CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Towards Chaos Engineering for Fault Injection Testing of Internal Automotive Systems

Quantifying Disturbance Tolerances in a Centralized Automotive Architecture

Master's thesis in Computer science and engineering

Elias Ekroth

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Towards Chaos Engineering for Fault Injection Testing of Internal Automotive Systems

Quantifying Disturbance Tolerances in a Centralized Automotive Architecture

Elias Ekroth

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2024

Towards Chaos Engineering for Fault Injection Testing of Internal Automotive Systems
Quantifying Disturbance Tolerances in a Centralized Automotive Architecture
Elias Ekroth

Supervisor: Vincenzo Gulisano, Department of Computer Science and Engineering
Advisor: Nikolaos Korkakakis, Volvo Cars
Examiner: Vincenzo Gulisano, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Towards Chaos Engineering for Fault Injection Testing of Internal Automotive Systems
Quantifying Disturbance Tolerances in a Centralized Automotive Architecture
Elias Ekroth
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

As the automotive industry becomes more and more reliant on software-defined functionality, the vehicle's internal communication system has to develop to keep up with the ever-increasing demands. A recent development is the centralization of the internal architecture, focusing most of the internal computation on one powerful central computer as opposed to across several small ones distributed throughout the vehicle. This introduces a single point of failure into the system and while automotive systems are built to be robust, the potential effects of such a failure should be investigated preemptively. To face this challenge, this thesis investigates a method to enable Chaos Engineering - negative testing in the production environment - in the automotive domain.

A Device-In-The-Middle fault injection system was developed and implemented into the core automotive system along with several fault models, enabling the disturbance of traffic flowing between the vehicle control unit and its connected gateway units, to quantify the disturbance tolerances of the system. Additionally, the throughput of the Device-In-The-Middle when exposed to increasing data rates was measured and compared to data rates expected of a vehicle in operation.

By systematically applying different disturbance magnitudes to repeated test case executions aimed at validating the core system, the system's approximate disturbance tolerances, along with some deviations from expected system operation, were found and analyzed. The combined tolerance and performance results indicate that, with some further development and latency optimization, the system has the potential to work as a chaos testing method in the automotive software testing process.

Keywords: chaos engineering, automotive, verification, validation, testing, negative testing

Acknowledgements

First, I would like to thank my industry supervisors, Nikolaos Korkakakis and Ivonne Nieto Granados, for their unwavering commitment and support to this thesis. I would also like to extend my sincere thanks to my academic supervisor Vincenzo Massimiliano Gulisano for his valuable feedback on my many thesis revisions. Next, I would like to thank the engineers at Volvo Cars who gave valuable insights into their system and its internal workings.

Additionally, I would like to thank Sindhuja Ramamoorthy of the Technical University of Denmark (DTU), who created the system utilized in this thesis alongside me. Our cooperation brought improvement and enjoyment to this thesis process.

Last but not least, I would like to thank my family, my friends, and my girlfriend for their kind words and support during this project.

Elias Ekroth, Gothenburg, 2024-11-03

Contents

List of Acronyms x

List of Figures xi

List of Tables xii

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Chaos Engineering . . . 2
- 1.3 Problem statement . . . 3
- 1.4 Ethical considerations and risk analysis . . . 4
- 1.5 Thesis structure . . . 4

2 Preliminaries 5
- 2.1 Volvo Cars' SPA2 platform . . . 5
- 2.2 Core System and Volvo Testing Framework . . . 6
- 2.3 Technology and Software . . . 7
  - 2.3.1 Media Converters . . . 7
  - 2.3.2 Iperf3 . . . 7
  - 2.3.3 tc-netem . . . 8

3 Problem specification 9
- 3.1 Problem definition and motivation . . . 9
- 3.2 Overview of proposed solution . . . 9
- 3.3 Research Questions . . . 11

4 Methods and Implementation 12
- 4.1 Device-In-The-Middle design . . . 12
  - 4.1.1 Disturbance models . . . 13
- 4.2 Device-In-The-Middle implementation . . . 15
  - 4.2.1 Traffic Forwarder Module . . . 15
    - 4.2.1.1 Traffic forwarding process . . . 16
    - 4.2.1.2 Disturbance implementation . . . 17
  - 4.2.2 Controller Module . . . 18
    - 4.2.2.1 Tolerance search algorithm . . . 18
    - 4.2.2.2 Results storage and visualization . . . 20

4.3 Applying the solution to the Core System . . . 21
4.3.1 Tolerance measurement . . . 21
4.3.1.1 Diagnostics test cases . . . 22
4.3.1.2 Software update test case . . . 23
4.3.2 Forwarder module variants for performance comparison . . . 23
4.3.2.1 C Traffic Forwarder . . . 24
4.3.2.2 Bridge interface Traffic Forwarder . . . 24
4.3.3 Performance testing setup . . . 25

5 Results **27**
5.1 Disturbance tolerance measurements . . . 27
5.1.1 Diagnostics tests . . . 28
5.1.1.1 Frame Delay . . . 28
5.1.1.2 Frame Loss . . . 29
5.1.1.3 Frame Bursts and Reordering . . . 30
5.1.1.4 Bit Error Rate . . . 31
5.1.1.5 Frame Duplication . . . 32
5.1.2 Software update test . . . 32
5.2 Number of unique errors found . . . 34
5.3 Observed system interactions . . . 35
5.3.1 Unresponsive unit after disturbed software update . . . 35
5.3.2 Increased test execution time . . . 35
5.4 Performance . . . 37
5.4.1 Device-In-The-Middle Bandwidth . . . 37
5.4.2 Device-In-The-Middle Latency . . . 39

6 Discussion **40**
6.1 Tolerance stability . . . 40
6.1.1 Diagnostics test cases . . . 40
6.1.2 Software update test . . . 41
6.2 Deviations from expected system operation . . . 42
6.3 Benefits to the existing testing system . . . 43
6.4 Implementation considerations . . . 44
6.4.1 Tolerance search algorithm . . . 44
6.4.2 Hardware and software implementations . . . 45
6.4.3 Performance measurements . . . 45
6.5 Limitations . . . 46
6.6 Future work . . . 46

7 Related work **47**
7.1 Automotive fault injection . . . 47
7.2 Chaos Engineering framework architecture . . . 48

8 Conclusion **49**

Bibliography **51**

List of Acronyms

VCU Vehicle Control Unit

VIU Vehicle Integration Unit

ECU Electronic Control Unit

CS Core System

VTF Volvo Test Framework

TS Test System

List of Figures

2.1 An overview of Volvo Cars' SPA2 platform E/E architecture [19]. . . 5
2.2 Logical topology of the Core System (CS) connected to the Volvo Test Framework (VTF), forming the Test System (TS). 6

3.1 Logical topology of the TS with the DITM between VCU and the connected VIU 10

4.1 Overview of the interaction between the DITM software modules . . . 12
4.2 The workflow of the forwarder module. FD indicates both File Descriptors of the sockets bound to the DITM interfaces. 16
4.3 Search algorithm process for the Delay disturbance applied to a random test. The green lines show the transition between the search algorithm phases. 20
4.4 The physical implementation of the DITM in the TS. 22

5.1 Differing Frame Loss tolerances between test cases C and D. 29
5.2 Tolerance search for frame duplication disturbance shown to never fail for test case G. 32
5.3 Number of unique error codes recorded per disturbance across the tolerance measurements. 34
5.4 Increased test runtime for test case F when exposed to high Delay disturbance magnitudes. 36
5.5 Vastly increased test runtimes for test case F using the Frame Loss and Burst Timeout disturbances. 37
5.6 Performance testing results for the hardware and software setups compared to performance without DITM. 38

List of Tables

4.1 Descriptions of the modeled disturbances. 14
4.2 The disturbances along with the parameters used for the tolerance measurements. 19
4.3 Descriptions of test cases used for system tolerance measurement. . . 23

5.1 Tolerance measurements for all tests using the Delay disturbance. . . 28
5.2 Tolerance numbers for all tests using the Frame Loss disturbance. . . 29
5.3 Tolerance numbers for all tests using the Burst Queue disturbance. . 30
5.4 Tolerance numbers for all tests using the Burst Timeout disturbance. 30
5.5 Tolerance numbers for all tests using the Frame Reordering disturbance. 31
5.6 Tolerance numbers for all tests using the Bit Error Rate disturbance. 31
5.7 Software update test tolerance results for each disturbance. 33
5.8 DITM Latency experiment results, shown as *min* / *avg* / *max* (ms). . 39

1

Introduction

In this chapter, the background of this thesis and an overview of the chaos engineering method are given first, followed by the problem statement and ethical considerations. Finally, an outline of the contents of the chapters is presented.

Since the 1980s, the amount of embedded software in road vehicles has steadily increased, with multitudes of passive and active features now being entirely software-controlled and consisting of millions of lines of code [1]. Among these features are safety-critical driver safety and assistance systems, such as cruise control, lane departure avoidance, and automatic braking systems [2], [3], requiring rigorous and regular testing and verification to ensure that these criteria are still met through software and hardware updates.

General testing of software functions is often done through *positive testing*, where valid data is input to the system to validate correct system behavior, but the prevalence of such testing has been shown to introduce a confirmation bias into the testing process [4]. A common recommendation is to additionally conduct *negative testing*, which entails testing through the use of invalid system inputs [5], [6], investigating the system's response to these types of stimuli. This kind of testing is recommended in the automotive industry through the *ISO 26262* standard for functional safety in road vehicles [7].

Recently, several actors in the automotive industry have begun moving from a decentralized internal system architecture, where vehicle functions are controlled by dedicated Electronic Control Units (ECUs) spread throughout the vehicle, to a centralized architecture, where a central computer controls these functions and sends orders to the connected ECUs. For example, Volvo [8], Aptiv [9], Microchip [10], and NXP [11] have all proposed centralized architectures for their vehicle systems. While this move presents new opportunities for functionality integration, it also presents new challenges for system reliability testing which may be faced using negative testing.

1.1 Background

The move to centralized in-vehicle systems comes with many benefits development-wise compared to decentralized architectures. These decentralized systems face challenges pertaining to e.g., scalability, performance of communication between internal

devices, and the cost of adding new ECUs to the vehicle system when implementing new functionality. Centralized architectures meet these challenges by centralizing the processing of functions to a single, highly performant computational device, connected to the rest of the vehicle's ECUs through gateway units that receive and forward orders from the central device. This change increases scalability and reduces development costs through easier inclusion of new software-defined vehicle functionality [12].

The greatest weakness of centralized architectures covered in the literature is the natural inclusion of a *single point of failure*, due to the centralization of functionality into one internal device. If this central unit should fail during operation, it runs the risk of total vehicle function failure, endangering the driver, passengers, and potentially other people around the vehicle [12]–[14]. To satisfy the ISO 26262 standard for functional safety, centralized systems must be constructed to be fail-operational, with the central computational unit showing redundancy for power, internal communication, and more [14].

While centralized automotive architectures facilitate the implementation of new functionality into the internal units, Volvo Cars wants to increase their understanding of how such functionality interacts with the vehicle's system in the case of unpredictable control unit behavior, specifically through faults in the communication domain. This type of negative testing is meant to simulate faults that may occur from a mis-configured or faulty device and measure how tolerant the system is to such disturbances. In this case, the *tolerance* is considered the average *magnitude*, i.e., the quantifiable size or strength, of a particular disturbance that is expected to stop the system from completing its intended function.

To address this challenge, Volvo Cars wants to investigate the introduction of Chaos Engineering to the automotive testing process. Chaos Engineering is a discipline in negative system testing that focuses on introducing unpredictable conditions into a system during operation and observing the system's behavior, increasing the understanding of the system's robustness under such conditions.

1.2 Chaos Engineering

While methods similar to Chaos Engineering have been used by several companies, such as Google and Meta (then Facebook), the term and the principles were formally introduced by Netflix in 2016 [15]. They sought to incentivize the design of robust and fault-tolerant systems through the development and use of tools like *Chaos Monkey*, which terminates random virtual machine instances on which the service is hosted [16]. The goal of this inclusion was to verify that the system degrades in a graceful manner even in the presence of unpredictable faults, and resulted in an increased focus on developing resilient systems within the company. The *chaos* element stems from the stochastic nature of the method, which aims to simulate the unpredictability of faults that may occur in system operation in a production environment.

The 4 principles of chaos engineering are as follows:

- Build a hypothesis around steady-state behavior
- Vary real-world events
- Run experiments in production
- Automate experiments to run continuously

A general chaos experiment is run by first defining the *steady-state* as some measure of the behavior of the system under normal circumstances. Then, a hypothesis that the steady-state will continue in both a *control group* and an *experiment group* is established. The experiment group is then subjected to events that aim to reflect real-world faults, which in the case of distributed systems could include events such as crashing servers, faulty network connections, and malfunctioning hardware. Finally, one tries to disprove the hypothesis by looking for differences between the control group and the experiment group.

An important part of the method is running the experiments in a production, or production-like, environment in order to accurately depict the unpredictable system behavior in the presence of faults, which opens new venues of research into improving system stability and resilience. Additionally, as systems change regularly through software and hardware updates, it is of equal importance that these tests are automated to run without user intervention.

Since its inception, Chaos Engineering has been applied to different kinds of systems. For instance, the *CloudStrike* and *ChaosXploit* tools used Chaos Engineering to evaluate the security of the Amazon Web Services infrastructure. Application of the former led to improved anomaly detection systems, and the latter successfully extracted information such as file extensions and user permissions [17], [18].

1.3 Problem statement

Volvo Cars wants to investigate the Chaos Engineering method as a guiding principle for improved software validation and verification of their centralized vehicle system through the simulation of unpredictable network conditions between internal devices. The ultimate goal of using this kind of negative testing is to improve the understanding of the disturbance tolerance of the system and to allow for the observation of system operation under the influence of faults.

This project aims to be the first step towards Chaos Engineering in automotive testing through the development of an extensible fault injection solution implementable in every stage of system testing. By leveraging the centralized architecture, the proposed solution consists of a computer device connected between the central controller unit and its connected devices, allowing for direct access to the traffic flowing between the devices. This enables the disturbance of said traffic through customizable means, providing an extensible and accessible access point to alter the data flow. By additionally allowing the computer device to interface with one of Volvo Cars' existing test frameworks, it can monitor the system response to disturbances during test case executions. This solution seeks to enable the measurement, anal-

ysis, and validation of the centralized vehicle architecture's tolerances to network disturbances such as latency, loss of data, and sudden changes in data rates.

1.4 Ethical considerations and risk analysis

Since several safety-critical vehicle features are controlled by software, proper testing of this software is vital to the safety of the driver and the passengers. Further expansion and improvement to this testing process stand to improve the reliability of the vehicle system while in use, thereby possibly reducing the number of accidents that occur as a result of the software of the vehicle itself. To attain this result, however, the testing process must be able to reliably identify issues before they are able to affect the customer. While this thesis aims to make modifications to the hardware in question, an additional aim is to affect the general system performance as little as possible, so that any discovered faults can in no way be attributed to the modifications themselves, ensuring that the results are useful in increasing safety. Additionally, the methods investigated in this thesis are tested in an early testing environment, instead of, for example, a complete vehicle driven by a human. This ensures that the method can be refined before potentially being used in more sensitive testing stages.

1.5 Thesis structure

Chapter 2 gives a deeper description of the centralized vehicle architecture and the testing setup available for this project, along with descriptions of specific hardware and software used. Chapter 3 deepens the motivation of the problem and gives an overview of the requirements and goals of the proposed solution, along with this thesis' research questions and limitations. In Chapter 4, the methods of this project are described, consisting of the design and implementation of the solution's system architecture, how the solution interfaces with the existing test framework to measure tolerances, and the method used to investigate the solution's viability in a full-scale vehicle system. Chapter 5 presents the results of the tolerance measurements and describes observed system responses, before presenting the viability results. The results, implementation, and potential future improvements are discussed in Chapter 6. The related works that this thesis drew inspiration from are presented in Chapter 7. Finally, the conclusions drawn from the project are presented in Chapter 8.

2

Preliminaries

In this chapter, more information pertaining to Volvo Cars' centralized architecture SPA2 is provided, along with a description of the testing setup utilized in this project. Additionally, the auxiliary hardware and software used in this project are presented. This is meant to provide an understanding of how the internal system architecture is connected, to more easily define the project parameters.

2.1 Volvo Cars' SPA2 platform

This project is conducted on the Scalable Product Architecture (SPA2) platform, an overview of which is shown in Figure 2.1. This platform encompasses the internal Electrical/Electronic (E/E) components of the new generation of Volvo Cars vehicles. It facilitates the implementation of infotainment, control, and safety systems, including engine control, proximity sensing, assisted steering, and braking systems.

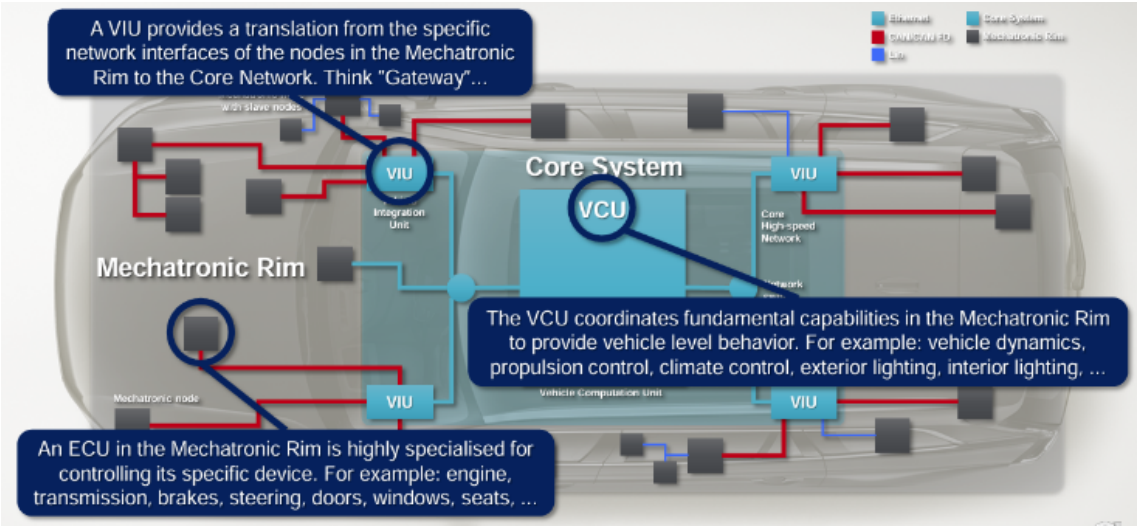


Figure 2.1: An overview of Volvo Cars' SPA2 platform E/E architecture [19].

SPA2 follows a centralized architecture, with three main types of internal units:

Vehicle Control Unit (VCU) The VCU is the central controlling unit of the vehicle. The responsibilities of the VCU include communication orchestration, vehicle and drive control, energy management, and diagnostics.

Vehicle Interface Units (VIUs) Also called Gateway Modules, VIUs hold the responsibility of interfacing between different sub-networks of the in-vehicle system. The VIUs act like highly performant switches that allow signals to traverse between the VCU and specific Electronic Control Units placed throughout the vehicle, and vice versa. The VIUs are connected to the VCU via 1000BASE-T1 Automotive Ethernet.

Electronic Control Units (ECUs) ECUs are small, specialized devices that control a specific vehicle function, such as brakes, engine thrust, and window control. Several of these devices are connected to each VIU. Several ECUs are connected to each VIU via CAN buses.

2.2 Core System and Volvo Testing Framework

As seen in the blue area of Figure 2.1, the *Core System* (CS) of the architecture consists of the vehicle's VCU and its connected VIUs. To test the core system in isolation, the ECUs are omitted and the core components are placed in a test rig, where controlled tests can be conducted, providing easy access to the devices and their connections.

The primary method of testing the CS is diagnostics tests, which instruct the CS to execute varying functionality and assert that the functions are completed. These *test cases* are handled by the *Volvo Test Framework* (VTF), an in-house developed testing framework for the CS. The VTF is housed on a computer physically connected to the VCU, allowing the VTF to execute test cases and monitor the results. This setup will be referred to as the Test System (TS), and the logical connections between the components are shown in Figure 2.2.

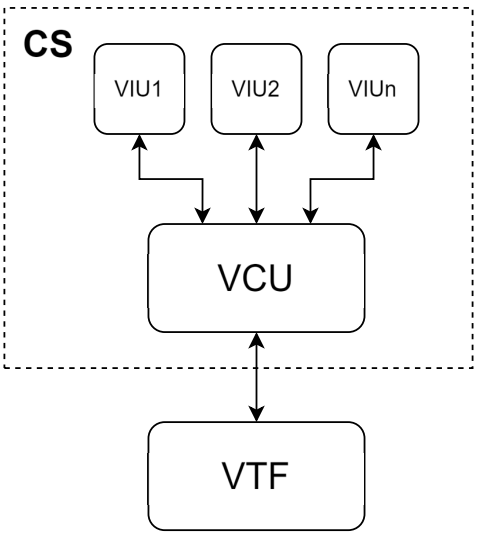


Figure 2.2: Logical topology of the Core System (CS) connected to the Volvo Test Framework (VTF), forming the Test System (TS).

A generalized VTF test case execution is as follows:

1. The VTF orders the CS to perform the selected functionality.
2. The VCU instructs the connected VIUs to perform this functionality, and the VIUs respond when finished. The VTF continuously monitors responses from the VCU and logs these.
3. The VTF asserts whether the test has passed or failed, and records the result. If the test case has failed, the VTF logs the reason for failure as an error code.
4. The test result and the continuous log are stored for later reference.

Examples of test case stimuli include reading the status of a unit, setting the unit to a certain internal mode, inducing a reset in the unit, and downloading software to the unit. Through such test cases, the VTF provides repeatable behaviors in the CS, validating its correct behavior. The VTF is used in this project to induce stimuli in the CS to enable tolerance measurement. The CS components constantly send some traffic between one another, such as heartbeat signals. This traffic persists during VTF test case executions.

2.3 Technology and Software

The hardware and software used during this project consist of Media Converters used to translate traffic from one medium to another, iPerf3 used for performance testing, and the tc-netem tool, used for traffic disturbance.

2.3.1 Media Converters

Media converters convert data signals from one physical medium to another, enabling communication between different types of networks. Automotive media converters mainly facilitate communication in the physical layer between devices using standard 100/1000BASE-T Ethernet interfaces with RJ-45 connectors and automotive Ethernet connections, such as 100/1000BASE-T1. These allow internal vehicular components to be connected to normal computers and are generally used to enable software and system testing. In this work, media converters are used to create an interface between the testing hardware and the VCU/VIUs of the vehicle test system.

2.3.2 Iperf3

Iperf3 is a Linux tool used to measure the maximum bandwidth over networks, supporting different parameters such as protocols (TCP, UDP, IPv4, IPv6...), transmitted packet sizes, and targeted bandwidth over the link. To test the bandwidth between two connected units, a server is started on the first unit, and a client connects to this server from the second, which starts the specified measurement. After measurement, the tool reports network information such as bandwidth, packet loss, jitter, and the number of transmitted packets [20].

2.3.3 tc-netem

The built-in Linux *tc* (Traffic Control) tool contains a *queuing discipline* called *netem* (Network Emulator). This command-line utility allows for the emulation of various network conditions on device network interfaces and is primarily used to evaluate the performance and robustness of network applications. The tool supports emulating network effects such as packet delay, loss, corruption, re-ordering, and duplication in an efficient manner [21].

3

Problem specification

In this chapter, the problem of the thesis is further explained, an overview of the proposed solution is given, and the research questions are stated.

3.1 Problem definition and motivation

This project aims to develop and investigate a fault injection method in the physical layer of the internal vehicular system, allowing for the use of the Chaos Engineering method for system validation and verification in centralized automotive architectures. The proposed solution is a hardware component that acts as a bridge between the VCU and its VIUs, forwarding and disturbing traffic between them through alterations such as delays, loss of traffic, and sudden decreases and increases in data traffic rate. The introduction of this kind of negative testing holds two purposes:

- The main purpose is to enable the quantification of *disturbance tolerances* for specific system functions. The tolerances denote disturbance magnitudes that the system can withstand before failing to successfully execute the function.
- The secondary purpose is the discovery of network disturbance magnitudes that induce substantial *deviations* from the expected system operation.

To demonstrate this solution, the fault injection method is tested on the Core System test rig at Volvo Cars, as shown in Section 2.

3.2 Overview of proposed solution

The proof-of-concept of this thesis involves the introduction of a *device-in-the-middle* (DITM) system to the CS testing. The DITM seeks to intercept and alter traffic flowing between the VCU and the VIU while they execute their specified functions, initiated by the VTF test cases. Through increasingly intense traffic disturbances, the unit behaviors can be observed. The purpose of this is to validate that the behaviors are consistent with the expected behaviors during such conditions and to find the tolerances of the test cases. These tolerances are indicated by the average disturbance magnitude that causes the test to fail.

The DITM functionality is designed around a Linux-based operating system on a device with multiple physical network interfaces, allowing for easy interface config-

uration and the connection of several external devices. The DITM system logic is built in the *Python* programming language, and control over the system is designed as a command line tool, through which the tester can specify which test should be investigated with which disturbance. Upon selection, the tool should search for and attempt to validate the system tolerance under operation by automatically altering the disturbance magnitude for repeated test case executions.

For this project, the link to be disturbed by the DITM is the Ethernet connection between the VCU and its associated VIUs, and an overview of this design is illustrated in Figure 3.1. This placement allows the observation of the effects of data traffic alterations on the control functions of the VCU.

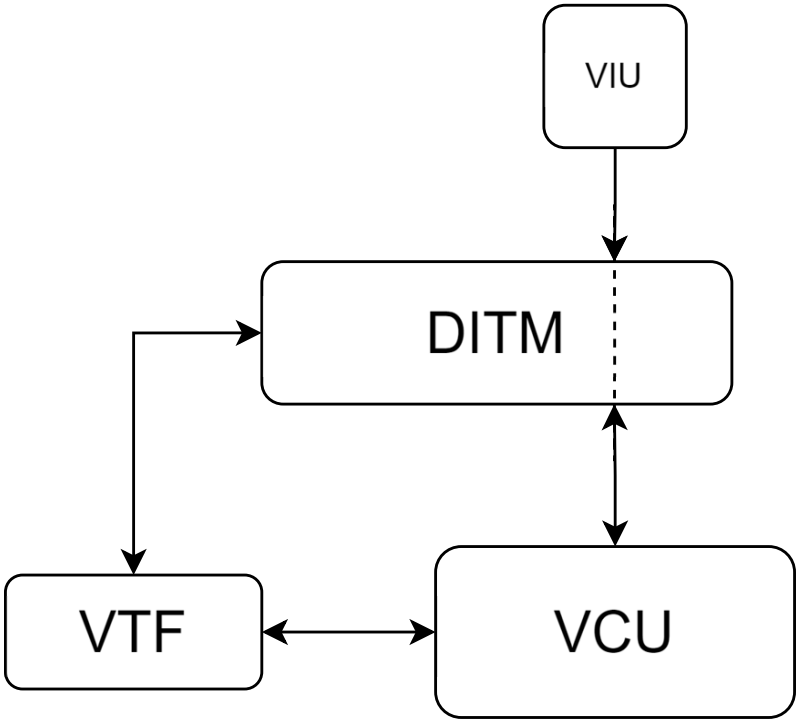


Figure 3.1: Logical topology of the TS with the DITM between VCU and the connected VIU

The VTF is used for system stimuli for the proof-of-concept DITM implementation. By running a test case at a disturbance magnitude of 0 and then systematically altering disturbance magnitudes throughout repeated test case executions, the following data points can be observed:

- System behavior during operation without disturbances, signifying the *steady state*
- System behavior during increasing disturbance magnitudes
- System behavior at test failure due to the disturbance in use
- System behavior during repeated application of disturbance magnitudes that caused a test failure

The end goal of this method is to validate that the system under test can repeatedly perform its intended function, despite repeated exposure to intense traffic alterations. By tracking runtime details of the system during the test runs, such as test execution time, disturbance magnitudes, and error codes, an additional aim is to observe potentially unintended or unexpected interactions between the devices under operation, opening avenues of further investigation into these interactions.

As the available tests only consist of diagnostic functions, they provide limited functionality compared to normal system operation in a full-scale vehicle. This work is therefore considered a first step in the path toward implementation in a full system and serves as a demonstration of some of the observations that can be made from this kind of negative testing.

3.3 Research Questions

With the problem definition presented in Section 1.3 as a base, this thesis presents the following research questions:

Q_1 **Can the proposed testing method reliably measure and validate the system under test's tolerance to communication disturbances?**

To ensure the usefulness of the proposed method, constraints must be placed on the expected results. For this project, the primary metric of the measured tolerances will be *reproducibility* during repeated test runs. Finding reproducible tolerances also allows for the repeated analysis of system operation beyond this tolerance level. Therefore, the second research question is:

Q_2 **Can the proposed testing method discover deviations from the expected operation of the system under test?**

Testing beyond the system tolerance allows for observing its operation in these conditions, additionally allowing for the potential discovery of deviations from system function. Like the previous research question, discovered deviations must be sufficiently *reproducible*.

Q_3 **Can the proposed testing method benefit the verification and validation of the system under test?**

The capabilities of the proposed testing method should be compared to the capabilities of the current testing method of the system under test, to gauge any potential benefits to be gained from applying this method in general testing.

4

Methods and Implementation

This chapter presents the methods used to approach the problem presented in the previous section. This chapter covers the design and implementation of the DITM, the modeling of the network disturbances, and the approach for measuring requirement tolerances of the test cases induced by the VTF. Finally, an approach for measuring the performance of the DITM is explained.

4.1 Device-In-The-Middle design

The software design of the DITM requires several software modules that facilitate traffic forwarding and alteration, analysis of results, and control over system execution, to function together. The modules included in this design are the *Traffic Forwarder*, *Controller*, *Disturbance Library*, and *Results Storage*. Figure 4.1 visualizes the modules' interaction.

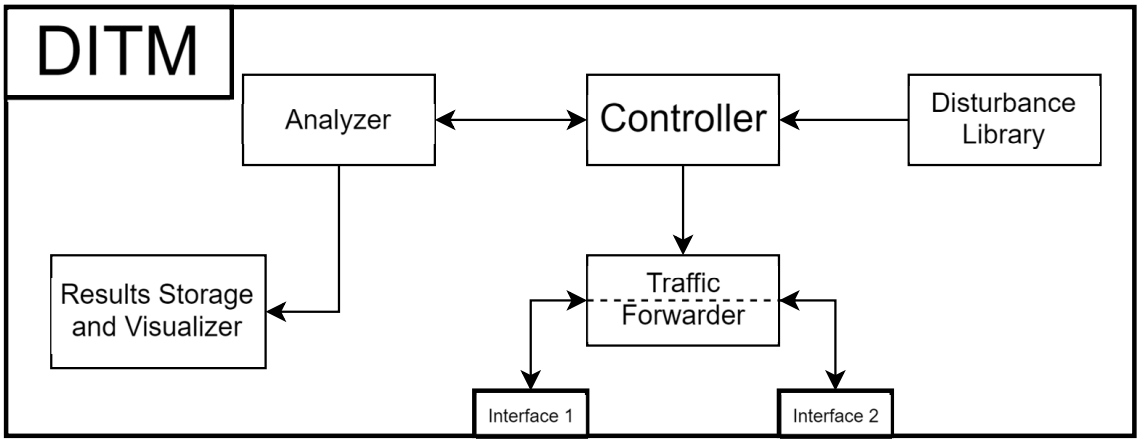


Figure 4.1: Overview of the interaction between the DITM software modules

The **Traffic Forwarder** provides the core functionality of the DITM, allowing network traffic to flow between the VCU and VIU without requiring reconfiguration of the CS. This enables the DITM to act as an invisible proxy for the network traffic. By physically connecting the two units to network interfaces on the DITM, these interfaces can be configured to forward traffic between each other.

The primary challenge for the forwarding module involves facilitating the alteration of the intercepted traffic in a desired and efficient manner during system runtime. Additionally, to ease the addition of new disturbance logic to the DITM, the logic should either be able to be written in Python or be implemented using existing Linux tools. Finally, there is the risk of slowing traffic between the units when altering the connection between them, as traffic has to go through more logic and connections than during transmission through a direct connection. Thus, the forwarding module must attempt to reduce this latency as much as possible, while still fulfilling the other requirements. This module receives instructions on when to forward traffic and which disturbances to apply from the Controller.

The **Controller module** is at the center of the DITM design, holding three primary responsibilities:

- Interfacing with the VTF to start test case executions.
- Coordinating the activation and magnitude of the selected disturbance, the parameters of which are fetched from the **Disturbance Library**.
- Recording and storing data points from the test case executions.

The data points of interest for the Controller Module are the *test runtime*, *test result*, and the potential *test error code*, as these data points allow for the monitoring of the general impact that the disturbances have on the TS during testing. To record the results of the test run, the controller utilizes the **Analyzer** module, which includes logic to read the log files generated by the VTF and extract the relevant data points, which are then given to the controller. The results of the tests are then transferred to the Results Storage.

The **Results Storage**'s responsibility is to efficiently structure and store the data points collected from the test runs for further analysis. To facilitate this analysis, the storage has to provide statistics of the number of test runs completed with each disturbance along with how many and which errors the disturbances triggered, and the ability to visualize the test run results. Finally, the storage must provide the user with the functionality to find the logs of specific test runs from the recorded data. This allows, for instance, backtracking from a certain error code and disturbance parameter to the test log, allowing for a deeper analysis of how the disturbance affected the test run.

4.1.1 Disturbance models

To demonstrate the functionality of the fault injection on the TS, seven disturbance types are modeled and implemented on the DITM, as described in Table 4.1. The disturbances are modeled as traffic delay, loss of frames, increases and decreases to data rate, alteration of sent frame order, a simulation of increased probability of bits being flipped in transmitted frames, and frame duplication. The table also describes the variability of the disturbances, i.e., how the disturbance magnitude is changed.

Disturbance	**Description**	**Variability**
Frame Delay	A communication delay put on the intercepted frame.	Per frame **delay**.
Frame Loss	The entire intercepted frame is lost.	Per frame **probability** of loss. The probability distribution is uniform.
Frame Burst - Queue	Strong intermittent increases and decreases in traffic rate.	**Threshold** before frames are sent as burst, represented as an integer **number of frames**.
Frame Burst - Timeout	Strong intermittent increases and decreases in traffic rate.	**Threshold** before frames are sent as burst, represented as a timeout.
Frame Reordering	Frames arriving in the opposite order to when they were sent.	Time frames are collected before being sent.
Bit Error Rate	Random bit flips in the bit stream.	**Probability** of flipping bits in the frame. The probability distribution is uniform.
Frame Duplication	Duplicate frames sent along the connection.	Per frame **probability** of duplication. The probability distribution is uniform.

Table 4.1: Descriptions of the modeled disturbances.

Frame delay and bursty traffic, referring to unexpected increases and decreases to the rate of traffic, can be caused by network congestion due to faulty switches or routers [22]. Delay is used to induce timeouts and simulate disconnected units, and can allow the monitoring of system performance with increasing response times in the vehicle network. As for bursts of traffic, to ensure correct operation, the hardware modules must be able to handle these sudden changes in traffic while still providing the correct functionality, and not allowing any important frames to be lost during the processing of previously arrived frames. The bursty effect is inspired by the microburst phenomenon which may occur in packet-switched networks [23]. The reason two variations of the Burst disturbance are implemented is to provide a view of both how many frames must be included in the burst before an effect is noticed on the system, and how long such a gathering of frames must happen, without having to save such metadata in a combined way.

Loss of data, bit flips, and frame duplication can be caused by errors in packet-based asynchronous data transmissions [24]. Frame loss is used to ascertain how system function is impacted when frames are lost with varying probabilities during all points of function execution. Simulating an increased bit-error rate facilitates

the investigation of system resilience to such a disturbance without requiring the use of radiation testing. Frame duplication makes sure that e.g., duplicate responses to messages don't induce duplicate function executions or unexpected statuses from affected units.

Frames arriving in the wrong order than sent can be caused by load balancing processes and system network loops. Simulating this allows for the validation that the sending and receiving order of the packets does not have an unexpected effect on the system or reduce system performance.

4.2 Device-In-The-Middle implementation

With the design as a base, the implementation of the DITM functionality starts with the basic functionality of the Traffic Forwarder. Then, disturbance logic is defined and tested, and both functionalities are combined to form the basic Controller Module. Finally, logic is inserted into the Controller Module to run test cases from the VTF framework and analyze the results, and a tolerance search algorithm is defined for the system under test.

4.2.1 Traffic Forwarder Module

Investigation into traffic forwarding starts with the open-source *Scapy* packet manipulation tool. Scapy is a Python tool that allows network traffic to be captured, decoded, forged, and sent along the connection, among other utilities, and is often used in the automotive industry due to its support for automotive communication protocols [25]. As such, the tool natively has interface bridging and packet sniffing functionality, such as the *bridge_and_sniff()* function, making it useful for this investigation. However, early testing showed that the use of this function incurred severe fluctuations in response time between interconnected units, stemming from the packet-analysis functionality of the function, slowing down the traffic forwarding by up to 13ms per packet. To improve this performance, Scapy's socket-creation functionality was utilized instead.

Scapy includes functions to create sockets and bind them to the connected interfaces on the DITM, allowing traffic to be monitored and transmitted between these sockets. The Scapy library natively includes socket creation through the *SuperSocket* class, allowing sockets to be created on different layers of the OSI model. For the purpose of this work, initializing layer 2, or link layer, sockets is the most fitting solution. Such socket configuration is done through the *scapy.conf* package.

A socket can be bound to an interface by calling the *conf.L2socket()* function, which works by calling on the *socket* Python library, which handles socket creation and configuration. This socket creation works by creating a raw socket, setting the socket to *promiscuous mode* to allow for the interception of all packets on the interface, and binding it to the intended interface. On completion, this function returns a socket file descriptor, which can be used to send and receive data on the interface.

4.2.1.1 Traffic forwarding process

On Traffic Forwarder program start, one socket is created on each interface, allowing access to the incoming data traffic. To detect when data is available on the sockets, the Python *select.select()* function is used, which monitors several socket file descriptors simultaneously. This function continuously polls the file descriptors in a blocking manner, only proceeding when one of the file descriptors has data available for reading.

Upon detection, the program calls the *handler function*, chosen on program start, which receives the data from the selected interface and saves it into a variable, to act as a buffer. Manipulating this buffer becomes the basis for implementing custom disturbances in the traffic, as it gives direct access to the data. Since the reception happens on the Ethernet level, the unit of data received is *Ethernet frames*. The entire traffic forwarding process is shown in Figure 4.2.

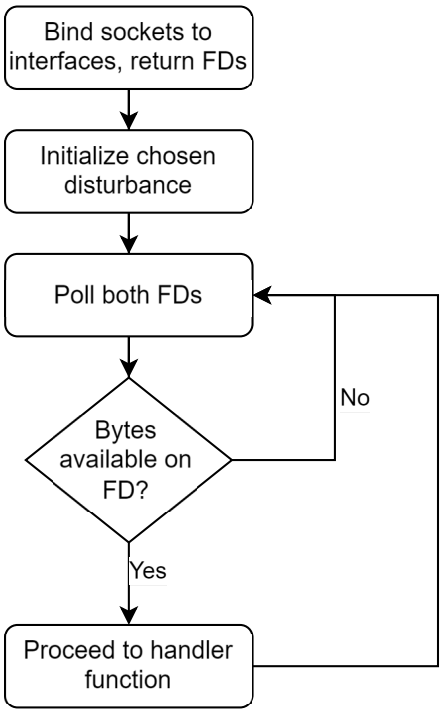


Figure 4.2: The workflow of the forwarder module. FD indicates both File Descriptors of the sockets bound to the DITM interfaces.

The basic handler function allowing for traffic forwarding takes the socket file descriptors of the source and destination interfaces as arguments, receives data on the source socket, and uses the destination socket to forward the frame on the wire. When data is received, it is temporarily stored in a buffer with a maximum size of 1500 bytes, since this denotes the maximum size of an Ethernet frame, excluding the header size.

This implementation provides ample opportunity to alter the data and its flow by defining additional handler functions that change the forwarding logic. This allows

for the implementation of custom data disturbances, to supplement the already existing Linux tools. The methods of implementing disturbances in the DITM are covered in the next section.

4.2.1.2 Disturbance implementation

As mentioned, the implementation of the disturbances is done through the handler functions of the Traffic Forwarder as well as the Linux tc tool. As shown in Figure 4.2, the chosen disturbances are initialized when the program starts, defining all variables used during runtime by the disturbance. Initializing these variables before the main loop of the forwarder program ensures that the number of expensive operations during runtime is kept to a minimum. After the initialization, the variables are ready to be used in the forwarder's sending and receiving loop through the handler function. When called, this function decides whether or not to send the frame(s) to the destination socket based on the properties of the chosen disturbance type.

While most early disturbance implementations were made using the tc-netem tool, continued development led to the move to Python scripts for most of the implementations. In the end, only the delay disturbance is implemented using tc-netem, as this was considered the most efficient implementation. Applying the Delay disturbance involves adding the chosen interface to the tc queuing discipline *netem*, allowing for network emulation options on that interface. Applying the Delay disturbance involves adding the target DITM interface to the *netem* queuing discipline of the *tc* tool. By then configuring the network emulation settings for the desired delay, a constant latency is placed on every frame sent through this interface. The magnitude of the delay disturbance is the *latency*, represented in milliseconds.

The Burst Queue, Burst Timeout, and Frame Reordering disturbances work using similar logic to one another. All three disturbances gather intercepted frames into an array-type data structure until a threshold is reached, after which all frames are sent as quickly as possible. While the disturbance is active, this threshold is checked each time a frame is intercepted by the DITM. While the Burst Queue and Burst Timeout use a Python list to gather frames, Frame Reordering involves the use of a Python *queue* instead, allowing for the frames to be extracted in a LIFO manner. The magnitude of the Burst Queue disturbance is the *number of frames* gathered before sending, while the magnitude of the Burst Timeout and the Frame Reordering is the *time* that frames are gathered before sending, represented in milliseconds.

For the Frame Drop and Frame Duplication disturbances, a random number between 0.0 and 1.0 is generated for each frame intercepted by the DITM and is compared to the chosen disturbance magnitude, the *drop probability*. The frame is either dropped or duplicated if the generated number is less than the disturbance magnitude.

For the Bit Error Rate disturbance, the disturbance magnitude is the chosen *probability* of a bit to be flipped, which is applied to every bit in the data stream. To make this disturbance efficient during runtime, a bit mask representing the flip probability is calculated at program startup by creating a long list of zeroes and then flipping each list element to 1 based on the flip probability. This bit mask can then be used to perform XOR operations on intercepted frames. When a frame n is intercepted

by the DITM, the disturbance performs an XOR of the bits of the frame and the first l_n bits of the bit mask, where l_n is the length of n in bits. When the next frame m is intercepted, the same operation is performed on the next l_m bits in the bit mask. When the end of the bit mask is reached, the operations start again from the beginning of the bit mask.

Seeds are used in the implementation of disturbances that rely on probability, i.e, Frame Loss, Frame Duplication, and Bit Error Rate. This ensures that the disturbance is applied consistently to the incoming traffic, given a certain probability, making the traffic the only truly random variable of a test, which increases the reproducibility of any conducted experiments.

4.2.2 Controller Module

The Controller module interfaces with the VTF and controls the Traffic Forwarding module through Python's *submodule* functionality, allowing commands to be run on the DITM as a new process. The controller is also responsible for finding the correct disturbance to utilize while making sure not to run the selected test case if no disturbance matching the input parameters is found. If this check passes, the tolerance measurement begins.

4.2.2.1 Tolerance search algorithm

In the first step of the tolerance measurement process, the logging process is initialized, creating file directories where the disturbance results will be stored. To allow for efficient storage of many data points, the Python package *pandas* [26] is used to create *DataFrames*, which allows for efficient storage, extraction, and manipulation of data. The content saved into the DataFrames during execution is based on data points read from the logs of the VTF by the Analysis module.

Secondly, the controller input arguments are used to fetch the correct disturbance type from the disturbance library. In this library, the classes of the disturbances contain important variables used to define and control the disturbance behavior, as listed in Table 4.2. In the table, P denotes the probability of the disturbance effect, while Q denotes the disturbance *quantum*. This quantum value is used in the tolerance search to signal when a tolerance has been found, as described below. For this work, the quantum values were chosen through manual experiments, where they are small disturbance magnitudes for which the diagnostics test cases, detailed in Section 4.3.1.1 could successfully run and complete. Similarly, the Max values for each disturbance that are not based on probability were chosen to be arbitrarily high, while the disturbances based on probability were chosen to be the natural upper bound of 100%.

Disturbance	Unit	Initial Value	Q	Min	Max
Frame Delay	ms	0	10	0	5000
Frame Loss	P	0.0	0.01	0.0	1.0
Frame Burst - queue	$frames$	1	1	1	1000
Frame Burst - timeout	ms	0	10	0	5000
Frame Reordering	ms	0	10	0	5000
Bit Error Rate	P	0.0	0.00001	0.0	1.0
Frame Duplication	P	0.0	0.01	0.0	1.0

Table 4.2: The disturbances along with the parameters used for the tolerance measurements.

Thirdly, the VTF is used to repeatedly run the chosen test case on the system, using the chosen disturbance to search for the tolerance of the test stimulus. To find this tolerance, a generalized *tolerance search algorithm* is used. This algorithm is based on the *Galloping Search* algorithm for unbounded searching originally proposed by Bentley and Yao, which involves using an exponentially increasing value to find the upper and lower bounds for a binary search [27], the total computational complexity of which is proven to be $O(log(n))$ [28].

The tolerance search algorithm consists of three phases: *Exponential increase*, *Binary search*, and *Validation*.

1. Exponential increase The exponential increase phase is used to discover the lowest values where the chosen test encounters a failure, or until the maximum value of the chosen disturbance is reached. This phase continuously doubles the disturbance magnitude value for each test case execution until a failure is reached and saves the $last\_fail$, and $last\_work$ values. These values denote the disturbance magnitudes where the test reported a success and a failure, respectively.

2. Binary search To pinpoint the tolerance, the $last\_fail$ and $last\_work$ values are in this phase used as the initial upper and lower bounds of a binary search. In each test case execution, the middle value is calculated through

$$\frac{last\_work + nr\_of\_epochs \cdot quantum}{2}$$

where nr_of_epochs is the number of whole quanta between the two values and is used to run another test iteration. This process continues until the difference between $last\_fail$ and $last\_work$ is lower or equal to the disturbance quantum, at which point the initial tolerance of the stimulus is considered found. This also enables the use of the disturbance quantum to adjust the sought granularity of the disturbance tolerance measurement.

3. Validation During this final phase, the validity of the identified tolerance is investigated by running the test with both $last\_fail$ and $last\_work$ as disturbance magnitudes for a set number of iterations. The default number of iterations for this

phase is four. After this phase is completed, the tolerance of this run is considered found.

4.2.2.2 Results storage and visualization

During each tolerance search, the system completes one full test run before continuing to the next iteration with a new disturbance value. Recording these values gives an observation of the system behavior at that disturbance level, saved as a row in a Pandas DataFrame. Each row in such a DataFrame is referred to as the *Test Sequence Index* and is used in the tolerance measurement graphs to represent time along the X-axis. Aggregating these DataFrames into one large DataFrame allows for efficient storage and access to the saved data, along with numerous visualization options through Python's various graph plotting tools, such as *matplotlib* and *seaborn*. Visualization of the system's tolerance to a certain disturbance is efficiently described with a line plot, giving a concise visualization of the trend of the tolerance.

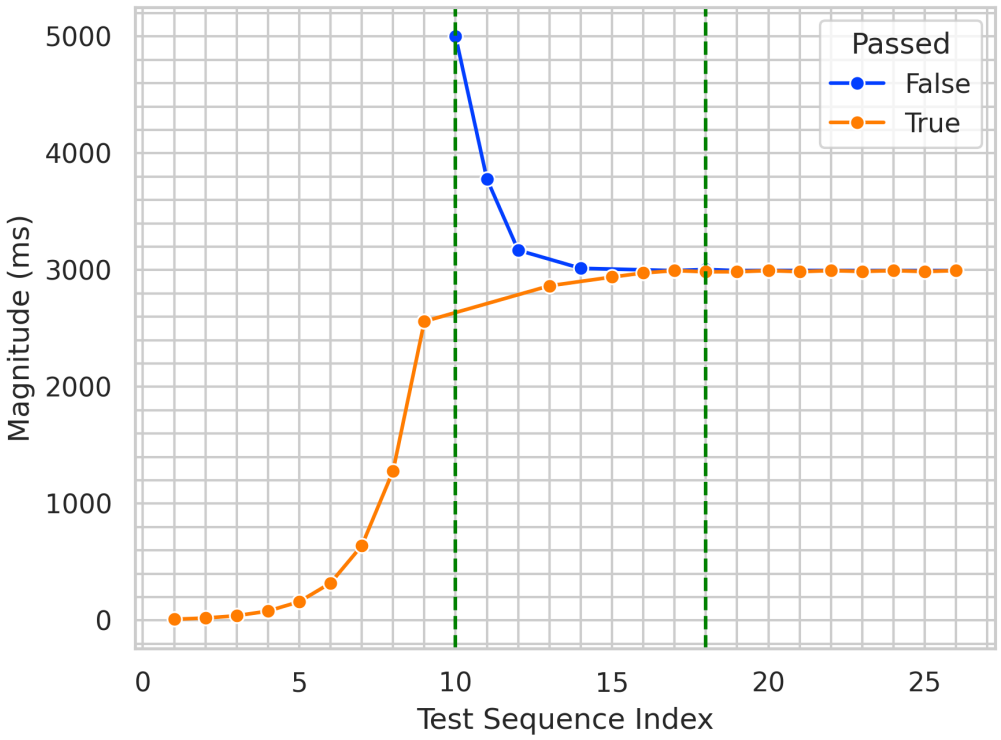


Figure 4.3: Search algorithm process for the Delay disturbance applied to a random test. The green lines show the transition between the search algorithm phases.

A tolerance measurement result is shown in Figure 4.3, where the Delay disturbance is applied to a test case checking the current status of the connected VIU. Each point in this graph indicates a full test case execution, indicated on the x-axis, with the dots showing the tests that passed and the blue dots showing tests that failed. The y-axis indicates the disturbance *Magnitude* used during the experiment. The left dashed line, at test execution 10, marks the test execution where a failure is detected, signifying the end of the Exponential Increase phase and the beginning

of the Binary Search phase. The right dashed line, at test execution 18, marks the next transition from the Binary Search phase to the Validation phase, where the measured tolerance is validated.

4.3 Applying the solution to the Core System

Two types of experiments are conducted to investigate the applicability of the DITM approach to testing the CS. The first type consists of disturbance tolerance measurements of the functions generated by test cases from the VTF. This type of testing investigates the proposed solution's ability to find the tolerances of specific functions, as well as potential deviations from expected execution, by employing the tolerance search algorithm introduced in Section 4.2.2.

The second type of experiment uses iPerf3 and the ping tool present in most operating systems to measure the overall Traffic Forwarder performance of the DITM compared to data volumes expected in a vehicle in operation, as well as giving a measurement of the latency incurred by the presence of the DITM. The performance testing aims to validate that the DITM can forward the volumes of traffic expected of a vehicle in operation, answering whether or not the DITM can be used in a full-scale vehicle system.

4.3.1 Tolerance measurement

The incorporation of the DITM system in the TS is shown in Figure 4.4. Here, the DITM system is implemented in the host computer running the VTF, with the DITM Controller Module configured to instruct the VTF to execute test cases on the CS. The VCU and VIU are connected to the host computer's two Ethernet ports via Media Converters (MC in the figure), with the traffic between the interfaces being connected and disturbed by the Forwarder Module.

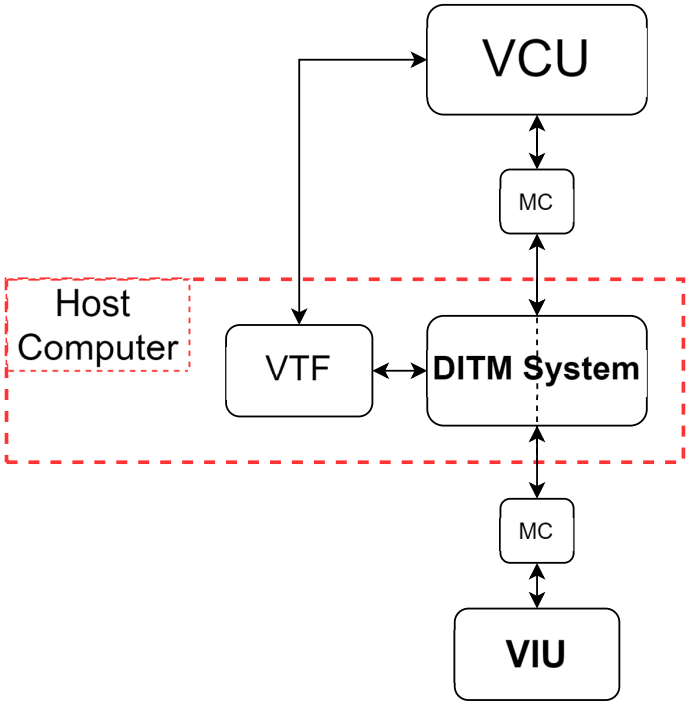


Figure 4.4: The physical implementation of the DITM in the TS.

The host computer employed for the tolerance measurements is an HP Z2 Mini G4 Workstation. The specifications of this hardware are further described in Section 4.3.2.

Existing VTF test cases are used to measure the disturbance tolerance of different system functions. The test cases are chosen due to their functions targeting the VIUs, thus allowing the DITM to intercept and disturb the traffic between the VCU and the targeted VIU.

The test cases can generate error codes, some of which are common between the tests. For instance, all test cases can generate a specific error code stating that a connection to the targeted device cannot be established. The traffic generated from the test cases all use the TCP protocol for communication between the units, providing some redundancy mechanisms such as re-sending packets that have not received responses within a certain time. This fact serves to further differentiate the test case stimuli from regular vehicle operation, which generally uses the UDP protocol for communication between internal devices.

The utilized test cases are split into two sets, due to differences in their effect on the targeted VIU. The first set contains 9 tests and is dubbed the set of *diagnostics test cases*, while the second set only contains one test, the *software update test case*.

4.3.1.1 Diagnostics test cases

The set of diagnostics test cases is described in Table 4.3, where the test cases are given a *test case designation*, in the form of a letter, and a description of their *effect*. The test case names and effects descriptions have been altered to avoid disclosing

Volvo Cars' internal information. The effects are therefore described in terms of which *method* the test uses and which *mode* the VIU is put in during the test, if applicable. The methods are dubbed *method*1 and *method*2, while the modes are called *mode*1, *mode*2, and *mode*3.

Test Case Designation	Test Case Effect
A	Read VIU part numbers
B	Read VIU current status
C	With *method*1, set VIU to *mode*2
D	With *method*2, set VIU to *mode*2
E	With *method*1, set VIU to *mode*3
F	With *method*2, set VIU to *mode*3
G	With *method*1, induce VIU reset in *mode*2
H	With *method*1, induce VIU reset in *mode*1
I	With *method*2, induce VIU reset in *mode*1

Table 4.3: Descriptions of test cases used for system tolerance measurement.

4.3.1.2 Software update test case

The software download process is an important part of the vehicle's lifecycle, allowing for important functional and security updates to the vehicle system. This process has historically been applied to vehicles during visits to workshops, but vehicle manufacturers have lately started employing these in Over-The-Air (OTA) updates instead, to improve customer satisfaction and enable quicker updates to e.g., safety and security requirements [29]. As these updates will impact important vehicle systems, the update process' response to failure and disturbance tolerance should be measured.

When the test case is executed, the VCU instructs the targeted VIU to download a software update from the VCU and apply it to itself. This update is sent as a stream of frames, and by disturbing this test the effect of e.g., lost, reordered, or delayed frames on the VIU's update process can be investigated.

4.3.2 Forwarder module variants for performance comparison

To test and compare the traffic forwarding performance of the DITM, the experimental setup shown in Figure 3.1 is altered into three different configurations used to test for performance differences when hardware with different specifications is used as the DITM.

In each configuration, the VIU is replaced with a Raspberry Pi 4 Model B, which is referred to as the *Edge Device*, to enable the use of iPerf3 to test the bandwidth. The hardware configurations subjected to performance tests are the following:

- Baseline: The Edge Device is connected directly to the VCU via a Media

Converter. This setup represents the setup with the least amount of hardware interference across the link.

- A Raspberry Pi 4 Model B configured as the DITM, connected between the VCU and the Edge Device.
- An HP Z2 Mini G4 Workstation configured as the DITM, connected between the VCU and the Edge Device.

The Raspberry Pi and the Workstation are used for this comparison due to their differences in:

- Processing power: The Raspberry Pi has a quad-core processor with a frequency of 1.8GHz, while the Workstation has an 8-core processor with a frequency of 3.8GHz.
- Power consumption: The Raspberry Pi requires a 15W power supply while the Workstation requires a 230W power supply.
- Price: A Raspberry Pi with 8GB RAM is currently sold for about 1000SEK, while the Workstation costs around 3000-4000SEK.
- Size: The Raspberry Pi occupies $19.5 * 88 * 58mm$ and weighs about 46 grams, while the Workstation occupies $58 * 215.9 * 215.9mm$ and weighs about 2.18Kg. This makes the Raspberry Pi much smaller and easier to transport and set up in different testbeds.

This makes the Raspberry Pi a more resource-efficient option, well-suited for scaling to every VIU in a full-scale vehicle, while the Workstation is the more performant option.

As mentioned in Section 3.2, the Python language is used to implement the DITM functionality due to its simplicity. However, this choice may also come with trade-offs pertaining to performance, as Python generally is slower than compiled languages such as C [30]. To investigate the potential differences in performance compared to alternative methods, two additional traffic forwarding approaches were created: A *C-implementation* approach and a *Bridge Interface* approach.

4.3.2.1 C Traffic Forwarder

A version of the Traffic Forwarder module was created, following the same forwarding process as described in Section 4.2.1.1. As C supports similar creation of and interaction with sockets as Python through the *sys/socket.h*, *sys/ioctl.h* and *sys/select.h* packages, the sockets can be created and monitored in the same manner.

4.3.2.2 Bridge interface Traffic Forwarder

This approach connects the DITM interfaces, dubbed *eth0* and *eth1* through a *bridge interface*, allowing traffic to flow between them. Bridge interfaces are generally used in networking to increase bandwidth, improve network performance, and provide

load-balancing capabilities. The commands used to create the bridge and connect the interfaces are shown in Listing 4.1.

```
1 ip link add name bridge0 type bridge
2 ip link set bridge0 up
3 ip link set eth0 master bridge0
4 ip link set eth1 master bridge0
5 iptables -A FORWARD -i bridge0 -o bridge0 -j ACCEPT
```

Listing 4.1: Commands used to set up bridge interface between Ethernet interfaces eth0 and eth1 on the DITM.

This is done by creating a new virtual interface of the *bridge* type and assigning it as the master of the existing interfaces. The traffic from the interfaces will then be forwarded to the *FORWARD* chain of *iptables*, the default Linux utility for defining packet filter rules. Since this chain has a default rule of *DROP*, setting this rule to *ACCEPT* will allow the traffic to flow between the interfaces freely.

4.3.3 Performance testing setup

To compare the possible performance differences, a version of the Traffic Forwarder module which is only capable of forwarding frames was implemented in the *C* programming language, due to its efficiency. The investigation aims to gauge the maximum traffic rate that can be forwarded through the DITM, using the *Interface Bridge*, *Python*, and *C* implementations on the *HP Workstation* and *Raspberry Pi* hardware setups, based on the amount of incurred packet loss at increasing traffic rates.

The data rate requirements for the Traffic Forwarder module have two thresholds: the *minimum* and the *recommended* rates. The minimum rate denotes the lowest rate the DITM should be able to handle to be usable, while the recommended rate is the overall target. The rates[1] are as follows:

- Minimum rate: 3 Mbits/s
- Recommended rate: 14 Mbits/s

The performance experiment setup modified the test setup by replacing the VIU with a Raspberry Pi capable of running iPerf3. The throughput between the VCU and the Raspberry Pi was tested by starting an Iperf3 server on the VCU and a client on the Raspberry Pi. The client command was formed as such:

```
1     iperf3 -c 10.0.0.1 -u -l <l> -b <BW>M
```

Note that the IP address listed is an example and is not a system address. This command sends UDP traffic from the server to the client at a bandwidth of *BW* megabits per second, with each packet sent being l bytes in length. The values of l

[1]The requirements are provided by Volvo Cars but do not represent actual internal requirements documentation.

tested are 64B and 128B[2], and the values of BW tested are between 5 and 50Mb/s, with increments of 5Mb/s between each step.

[2]Values given by Volvo Cars

5

Results

This section describes the results of applying the DITM, using the test cases described in Table 4.3. The results cover the tolerance measurement results, deviations from expected operation observed in the system during experimentation, and the performance measurement results of the three traffic forwarding module implementations.

5.1 Disturbance tolerance measurements

The search algorithm covered in Section 4.2.2.1 was used to repeatedly narrow down the tests' tolerances to the disturbances shown in Table 4.2. The tolerance search algorithm was allowed to run to completion 10 times for each *test case - disturbance* combination. In general, the Delay, Burst Timeout, Frame Reordering, Frame Loss, and Bit Error Rate disturbances required between 20-25 test case executions for the algorithm to terminate, while the Burst Queue and the Frame Duplication disturbances required between 10-15 test case executions. The means of all disturbance magnitudes measured during the algorithm's validation phases form the total tolerance trend, while the standard deviations of these measurements are used to show the tolerance stability.

Due to their similarities, the measurement results of the diagnostics tests will be presented together, while the software download function's results will be shown on their own.

One important aspect of these experiments is that these test cases generate lower amounts of traffic than what flows through a full-scale vehicle during operation. This was done due to limitations in the available testing setup and is discussed further in Section 6.5.

The test cases are modified to only target the VIU connected to the DITM, to test the unit in isolation. Since a faulty system generally involves a single malfunctioning unit, this investigation only simulates faults from the VCU. While some tests were run in the other direction as well, due to similarities between the results only the VCU to VIU direction was focused on.

5.1.1 Diagnostics tests

The diagnostics procedures are important utilities as they allow for the investigation of the hardware units' correct basic functions, statuses, and communication capabilities. Testing the fault tolerance of these functions gives an approximation of the risk those tests have of failing at the chosen disturbance level while evaluating how test failures affect the system.

For all test cases, the disturbance tolerances are measured using the HP Workstation DITM and Python software implementation. During this testing, the disturbances are applied to the DITM interface which faces the VIU, simulating disturbed traffic coming from the VCU.

5.1.1.1 Frame Delay

	Delay (ms)
Test Case	mean / σ
A	1995.0 / 5.017452
B	2991.883117 / 5.179078
C	1860.0 / 197.756649
D	1995.0 / 5.015699
E	1995.0 / 5.063697
F	1995.0 / 5.052912
G	903.0 / 10.177905
H	1995.0 / 5.063697
I	1995.0 / 5.013072

Table 5.1: Tolerance measurements for all tests using the Delay disturbance.

The Delay disturbance shows the most stable tolerance measurement results out of the disturbances, as shown by the results' low σ values, meaning that the results in the search algorithm's validation phase rarely differ from the mean.

While most tests show a clear tolerance level of around 1995ms of delay, the two clear outliers are test cases B and G. Investigation into this interaction shows that these differing values are caused by differences in code structure between the tests, making this behavior an expected part of system operation.

The other clear outlier is test case C, which shows a higher σ value than the other tests, which may indicate a rarer error. Investigation of the test logs shows that the test case usually fails at a delay magnitude of 1930ms, while there is one failure at a magnitude of 1280ms.

5.1.1.2 Frame Loss

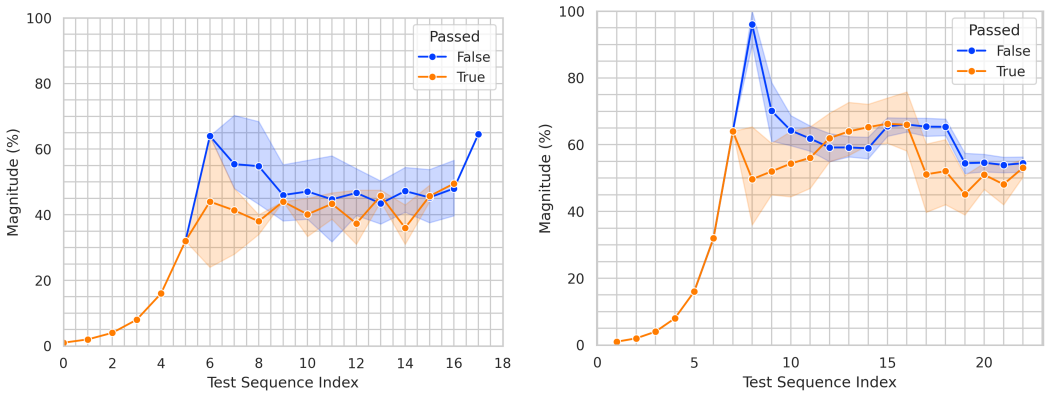
	Frame Loss Probability (%)
Test Case	mean / σ
A	57.0922 / 14.2219
B	58.8484 / 17.1991
C	43.8781 / 9.5381
D	59.6637 / 14.337
E	35.075 / 10.1277
F	60.5254 / 18.7197
G	37.252 / 13.2652
H	63.0273 /4.5222
I	53.873 / 13.3146

Table 5.2: Tolerance numbers for all tests using the Frame Loss disturbance.

The frame loss disturbance shows a high spread in the results, as seen in Table 5.2. While the results may seem to show that the system is very resistant to frame drops, the low amount of traffic generated by the test cases simply requires a high disturbance magnitude before a frame important to the diagnostic function is removed from the traffic.

Most test cases appear to have quite a high tolerance to frame loss, requiring disturbance magnitudes of around 50-60% before failing. However, test cases C, E, and G show remarkably lower tolerances, while keeping σ values comparable to the rest of the tests, indicating a lower tolerance to frame loss.

The aforementioned effect is visualized in Figure 5.1, showing the spread of the search algorithm process for test case C when compared to the more typical test case D. Notice that the first failure for test case C always occurs at a much lower disturbance magnitude than test case D. The clouds in the figure indicate the spread of repeated test case results throughout the repeated tolerance search algorithm executions.



(a) Test case C

(b) Test case D

Figure 5.1: Differing Frame Loss tolerances between test cases C and D.

5.1.1.3 Frame Bursts and Reordering

The frame burst and reordering disturbances all operate using similar logic, with the intercepted frames being put into a list before being sent all at once when a threshold is reached. As mentioned in Section 4.2.1.2, the disturbance threshold is checked each time a frame is intercepted by the DITM, blocking the sending of already received frames otherwise. In a larger-scale system with higher volumes of data flowing through the connection, this issue would be circumvented by the condition being checked very often. During the diagnostics tests, however, the data rate is low enough to interfere with this logic, causing great fluctuations in the tolerance measurements.

This behavior can be observed in the tolerance measurement results when using the **Burst Queue**, **Burst Timeout**, and **Frame Reordering** disturbances, as shown in Tables 5.3, 5.4, and 5.5. For the Burst Queue, this effect is predictable as the logic is based on the number of frames in the list, but the accuracy of the Burst Timeout and the Frame Reordering disturbances suffer due to this.

	Burst queue depth (frames)
Test Case	mean / σ
A	7.822581 / 2.008276
B	8.985294 / 4.631013
C	4.772727 / 1.008421
D	4.675 / 0.722939
E	5.9 / 1.836943
F	6.375 / 1.530394
G	2.5 / 0.503509
H	4.5 / 0.50274
I	6.527778 / 1.899372

Table 5.3: Tolerance numbers for all tests using the Burst Queue disturbance.

	Burst timeout (ms)
Test Case	mean / σ
A	2506.125 / 384.986298
B	4292.375 / 708.838174
C	751.0 / 1055.118902
D	1828.493151 / 390.280787
E	1685.0 / 5.063697
F	2056.477273 / 934.589034
G	220.0 / 8.473185
H	1887.5 / 435.571664
I	1842.875 / 263.044314

Table 5.4: Tolerance numbers for all tests using the Burst Timeout disturbance.

	Reordering timeout (ms)
Test Case	mean / σ
A	1848.75 / 149.063112
B	2590.25 / 756.699179
C	657.0 / 565.813234
D	1799.0 / 443.694042
E	1676.0 / 27.808871
F	1911.666667 / 392.926045
G	205.0 / 25.72039
H	1685.0 / 5.063697
I	1727.75 / 85.192715

Table 5.5: Tolerance numbers for all tests using the Frame Reordering disturbance.

What can be observed from these three results tables are patterns that also arise in the Delay disturbance tolerances. While test case B shows a generally higher mean tolerance to the disturbances, test cases C and G show generally lower tolerances. What this indicates is that the increased data rate from the burst disturbance does not affect the outcome of the test case, as the test failure likely occurs due to a timeout when the frames are in the disturbance queue.

5.1.1.4 Bit Error Rate

As the Bit Error Rate flips random bits in the data stream, random fields in the intercepted frames can potentially be changed with unknown side effects. However, no extraordinary events outside of the test failures could be observed while using this disturbance, indicating that corrupted frames were dropped when reaching their destination.

	Bit flip probability (%)
Test Case	mean / σ
A	0.0688 / 0.0287
B	0.1721 / 0.0757
C	0.0569 / 0.0244
D	0.0686 / 0.0285
E	0.0512 / 0.0132
F	0.0777 / 0.0448
G	0.0385 / 0.0235
H	0.0626 / 0.0323
I	0.0822 / 0.0246

Table 5.6: Tolerance numbers for all tests using the Bit Error Rate disturbance.

The results detailed in Table 5.6 show very high tolerances to bit error rates compared to expected values in embedded systems. Like the frame drop disturbance, this can be explained by the low amount of traffic flowing through the system during

test case execution, requiring higher disturbance magnitudes to cause the test cases to fail. The previously observed lower tolerances of test cases C, E, and G can also be observed in these results, strengthening the notion that these tests have higher probabilities of failure due to omitted data than the other tests.

5.1.1.5 Frame Duplication

Using the search algorithm with the frame duplication disturbance did not trigger any test failures, an example of which is shown in Figure 5.2. To further investigate the possibility of this disturbance affecting the system, the disturbance was altered to duplicate frames up to 5 times instead of just once. This, however, yielded the same results as the initial implementation.

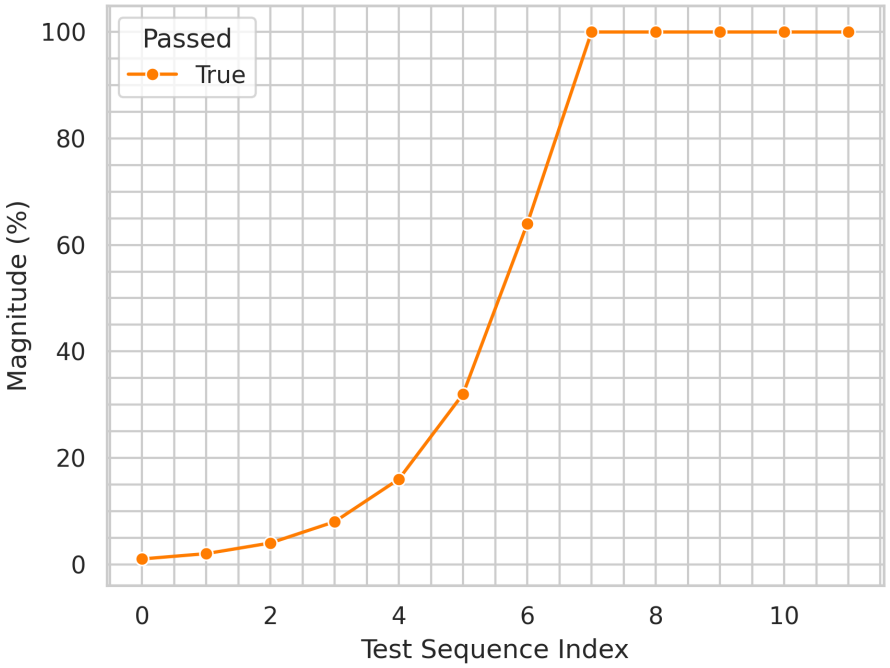


Figure 5.2: Tolerance search for frame duplication disturbance shown to never fail for test case G.

5.1.2 Software update test

It is observed that the software update test generates more traffic critical to successful function execution than the other tests, and several tolerance levels are subsequently lower than observed for the diagnostics tests. A device failure was also recorded during this testing, described in Section 5.3.1. Like with the diagnostics test cases, the Frame Duplication never caused the software update test case to fail.

Two changes to disturbance configurations and logic were required to measure the tolerance of this test properly. Firstly, the quantum of the Bit-Error Rate disturbance had to be lowered from 0.00001 to 0.00000001, as the test was never completed otherwise.

Secondly, the Burst Timeout and Frame Reordering disturbances always made the test fail at their initial disturbance values, which is unexpected as they should be more similar to the delay disturbance. To rectify this, the logic of the Traffic Forwarder module was changed to constantly check the burst and reorder time thresholds. While this bursts the frames at more accurate timings, it also requires the threshold to be checked more often, potentially decreasing overall performance when used with higher data rates.

The tolerance measurement results using the updated disturbances are shown in Table 5.7.

Disturbance	mean / σ
Delay (ms)	494.0 / 8.711913
Frame Loss (%)	2.3469 / 0.9751
Bit Error Rate (%)	1.661590e-06 / 0.9994573e-06
Burst Queue (frames)	2.3 / 0.656947
Burst Timeout (ms)	535.0 / 5.773503
Frame Reordering (ms)	10.333333 / 12.617806

Table 5.7: Software update test tolerance results for each disturbance.

As the software update test generates more traffic important to the outcome of the test case, the observation of lower tolerances when compared to the diagnostics tests is expected. The delay results are expectedly stable, and the Frame Drop disturbance results show greater stability compared to the diagnostics tests, but a greater exactness could likely be found by reducing the disturbance quantum value. As the Burst Timeout results are similar to the Delay results, the conclusion can be drawn that the software update process does not suffer from the increased data rate.

The most interesting results come from a comparison between the Burst Timeout and Frame Reordering disturbances. The disturbance logic functions similarly for both disturbances, yet the Frame Reordering tolerance is much lower than the Burst Timeout tolerance. When inspecting the test case logs, it is revealed that while the software update test can sometimes succeed with the disturbance active, the magnitude required to fail the test generally falls at 10ms of gathered frames, and the highest magnitude recorded for a successful test execution was 40ms. To investigate this further, the software update test was run while tracking how many frames were in the Frame Reordering queue when the threshold was reached. At the mean validation phase magnitude of 10ms, the max number of frames in the reordering queue was around 4-7 throughout the test case execution, indicating that even a low number of frames arriving out-of-order is enough to cause a test case failure. For comparison, the Burst Timeout disturbance showed an equal number of maximum frames in its list during test case executions.

5.2 Number of unique errors found

In addition to the tolerance measurements, the merit of the disturbances can be investigated by analyzing how many unique error codes each disturbance can find when used on the test cases. This gives an idea of how useful the disturbances can be in evaluating a function in-depth, by disturbing it at different stages during execution. These results are taken from the same test case executions described in the previous sections and are shown in Figure 5.3. The total number of unique error codes triggered during all test case executions is 21.

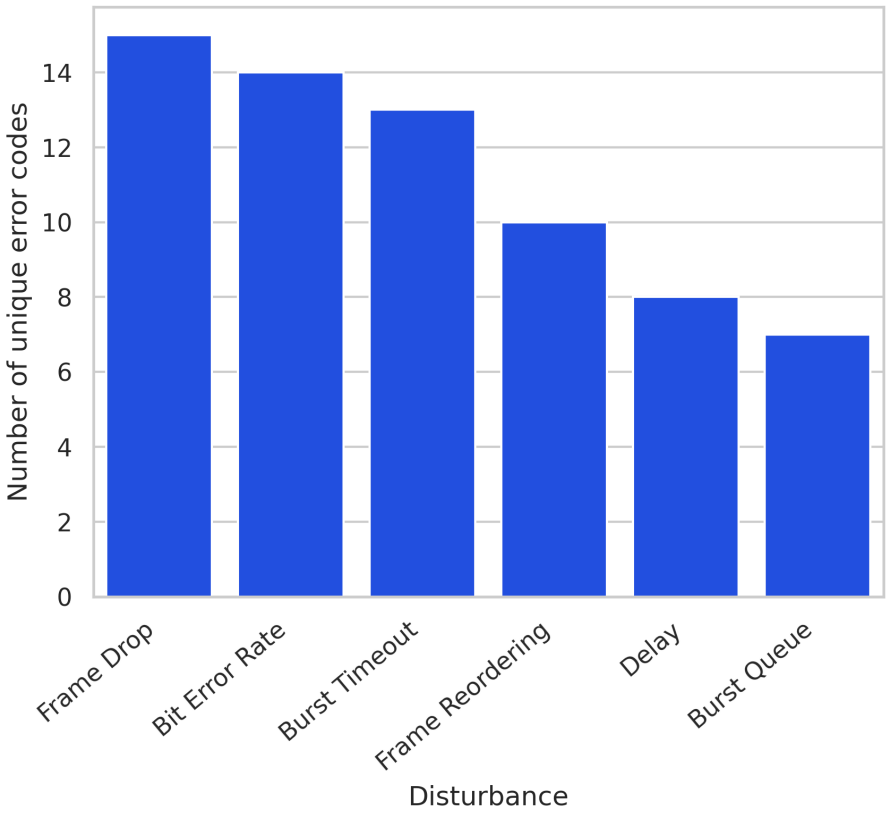


Figure 5.3: Number of unique error codes recorded per disturbance across the tolerance measurements.

The Frame Drop appears to be the most potent at triggering different errors, triggering 15 out of the 21 discovered, closely followed by the Bit Error Rate and the Burst Timeout disturbances. The Frame Reordering triggered fewer errors than the Burst Timeout, which is surprising considering the similarities between their implementations and indicates that the Burst Timeout may be slightly more adept at triggering errors in these kinds of tests. That Delay can trigger a greater number of errors than the Burst Queue is also somewhat surprising, due to the static nature of the Delay disturbance and the more fluent nature of the Burst Queue.

5.3 Observed system interactions

Some interesting interactions between the connected units were observed during the disturbance testing of the system under test. These interactions explain how the connected units and the test functions respond to the applied traffic disturbances.

5.3.1 Unresponsive unit after disturbed software update

A VIU unit was found to be unresponsive after a tolerance measurement of the software update test case using the Frame Loss disturbance. During the tolerance search, the test case was executed a total of 11 times, where the first 3 test executions were successes and the remaining 8 executions were failures. The first 6 failed test runs returned an error code stating that the download itself was unsuccessful, as is expected from the test. However, the two last test runs returned an error code indicating that no connection could be established to the connected unit, which is an atypical failure reaction for this function.

Connection to the unit was firstly tested in two ways: by pinging the VIU from the VCU, and by executing diagnostics test case A (check status), as described in Table 4.3, with both methods returning no responses. Following this, attempts were made to recover the VIU by running the software downloading test again without disturbances, both with and without the DITM connected. Neither of these methods succeeded in recovering the VIU's functionality. Additionally, external tools were used to attempt to reset the VIU software, which also failed, confirming that the VIU had been left in a broken state by the faulty execution of the test.

While this interaction may indicate a weakness in the software update process, the effect was not reproducible. This was tested by replacing the VIU and running the tolerance search algorithm with the Frame Loss disturbance a total of 10 times using the frame loss disturbance, during which the effect did not occur. This indicates an exceedingly rare fault in the software download process or a hardware issue with the first tested unit.

5.3.2 Increased test execution time

When the disturbance magnitudes were great enough to simulate a disconnected unit, e.g., through a sufficiently high delay value or 100% frame loss, the runtime of test case F was increased by a remarkable amount. As seen in Figure 5.4, the runtime of each test case execution scaled linearly from around 60 to 200 seconds with an increasing Delay magnitude, before increasing to around 1200 seconds on test failure when the Delay disturbance exceeds 2000ms.

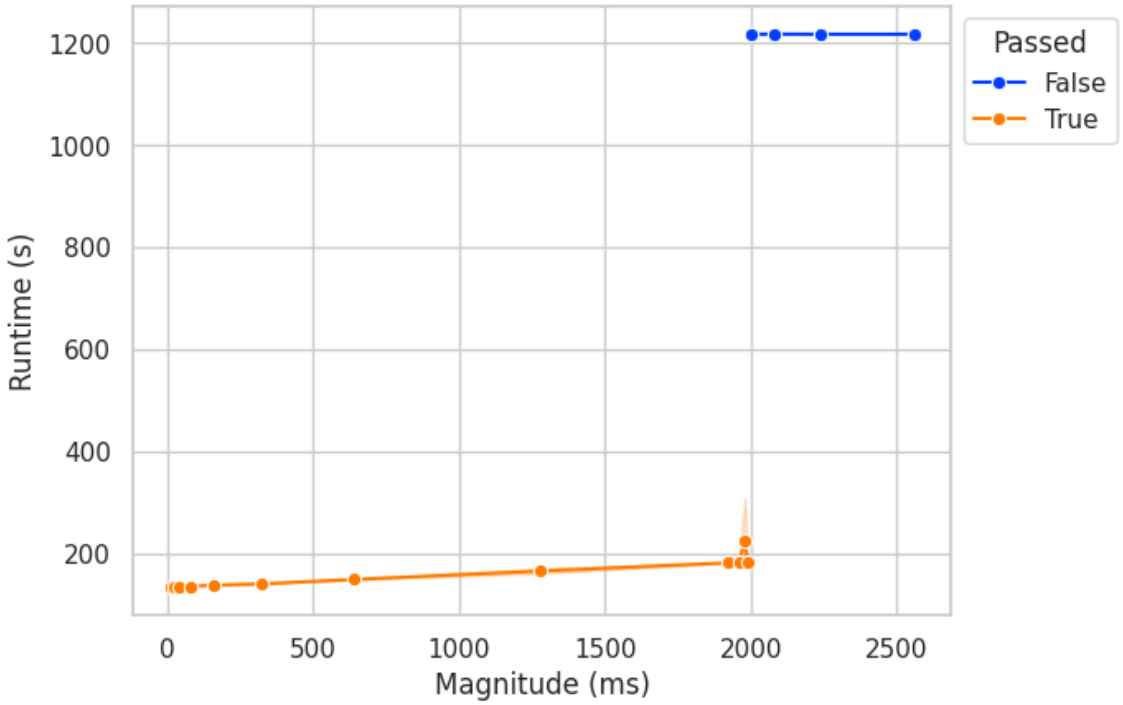
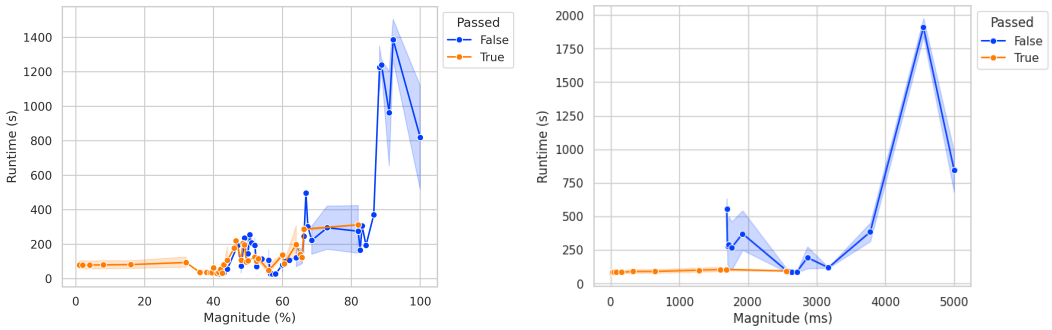


Figure 5.4: Increased test runtime for test case F when exposed to high Delay disturbance magnitudes.

Investigation into this interaction reveals that upon test failure, the VIUs undergo a reset and respond to the VCU upon reset completion. However, for each VIU that is reset, the VCU needs to communicate with all VIUs, incurring a significant timeout period if any VIU is non-responsive. As the process is run once for each connected VIU, the combined timeout periods give the failed test a massively increased completion time. This effect holds as long as the VIU cannot communicate with the VCU, meaning that high disturbance magnitudes that do not simulate total disconnection only increase the test completion time somewhat, as seen for the Frame Drop and Burst Timeout disturbances in Figure 5.5. These two disturbances are also shown to be able to increase the test runtime further, up to about 1900s per test execution in the case of Burst Timeout. This runtime-increasing effect was also observed for the Frame Reordering disturbance, although this disturbance did not increase the runtime as much.



(a) Frame Loss (b) Burst Timeout

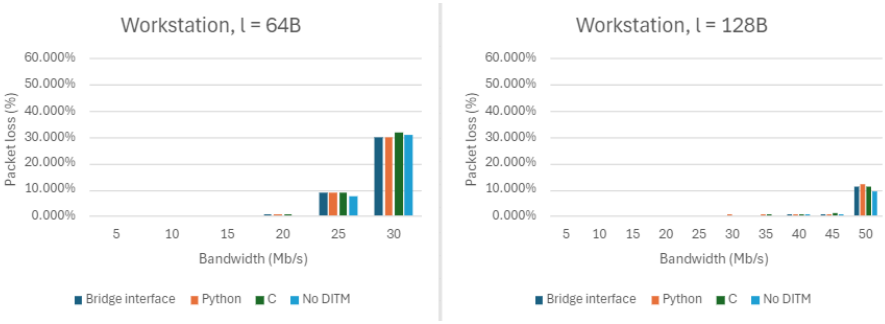
Figure 5.5: Vastly increased test runtimes for test case F using the Frame Loss and Burst Timeout disturbances.

5.4 Performance

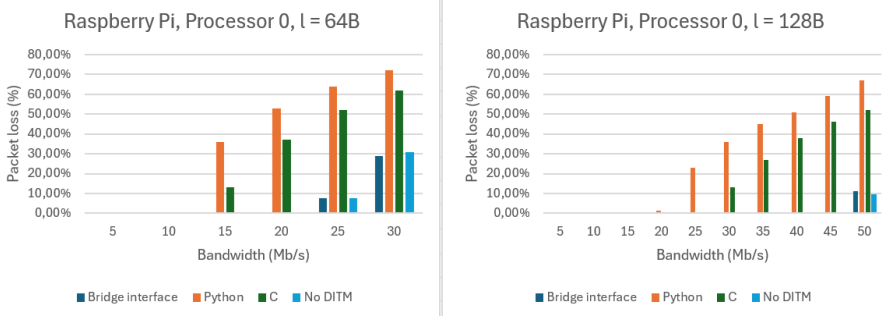
The performance of the three software implementations was tested using iPerf3 using both the HP workstation and the Raspberry Pi as the DITM. In addition, the latency added by the presence of the DITM was measured using the ping tool.

5.4.1 Device-In-The-Middle Bandwidth

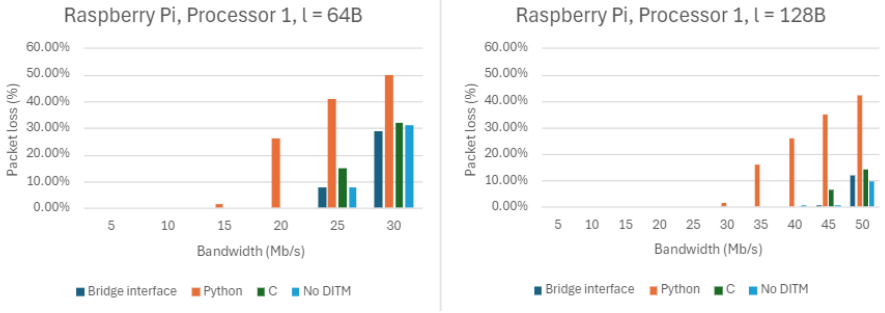
The bandwidth measurement results using the hardware and software setups described in Section 4.3.2 are shown in Figure 5.6.



(a) Workstation



(b) Raspberry Pi, Forwarder Module process set to processor 0



(c) Raspberry Pi, Forwarder Module set to processor 1

Figure 5.6: Performance testing results for the hardware and software setups compared to performance without DITM.

From these results, the impacts of differing hardware for the software implementations can be seen. For the workstation implementations, all three implementations for both l values performed as well as not having the DITM present. This is, however, likely due to limitations with the measurement setup, as discussed further in Section 6.

For the Raspberry Pi, it was observed that the performance was different depending on whether or not processor 0 was issued the Traffic Forwarder process during throughput testing, as processor 0 showed worse results. By setting this process affinity to a processor other than 0, in this case 1, the overall performance of the Raspberry Pi implementations increased. This is only applicable to the Python and C implementations of the Forwarder Module.

Overall, the Python implementation of the Forwarder Module shows worse through-

put on the Raspberry Pi than the C and Bridge Interface implementations, which both demonstrate throughput comparable to when using the Workstation, as long as processor 0 is assigned to the Forwarder Module process.

5.4.2 Device-In-The-Middle Latency

Using the same DITM setups as described in the previous section, the latency added to the communication between the VCU and the Edge Device is measured using the ping tool. The latency between the VCU and the Edge Device without the presence of the DITM is also measured, to serve as a point of comparison.

Each latency experiment is conducted by sending 5000 pings at a rate of 200 pings per second from the Edge Device to the VCU, with the minimum, average, and maximum response times being recorded. The results from these experiments are shown in Table 5.8, where they are shown as *min / avg / max*, and are measured in milliseconds.

For comparison, conducting this experiment while no DITM is connected results in *min / avg / max* values of 0.074 / 0.183 / 0.474.

Software	Workstation	Raspberry Pi
Python	0.372 / 1.221 / 3.635	0.428 / 1.358 / 4.044
C	0.407 / 1.203 / 3.078	0.572 / 1.354 / 3.158
Bridge	0.496 / 1.176 / 2.978	0.541 / 1.287 / 3.150

Table 5.8: DITM Latency experiment results, shown as *min / avg / max* (ms).

As shown by these results, the presence of the DITM appears to on average add 1-1.2ms to the response time between the VCU and the Edge Device. The maximum latency values shown in the result were generally observed for the first ping of the sequence, after which the latency stabilized.

6

Discussion

This chapter discusses challenges encountered during this work and the results of the thesis investigation. This discussion seeks to answer the research questions posed in Section 1.3. The discussion encompasses the observed *tolerance stability*, based mainly on the standard deviation values seen in Section 5, the *observed deviations from regular operation*, and the potential *benefits* to including this testing in the existing testing system. Afterward, challenges concerning the hardware and software implementation of the DITM approach are discussed, and improvements are suggested.

6.1 Tolerance stability

The first research question was *Can the proposed testing method reliably measure and validate the system under test's tolerance to communication disturbances?*. This question is answered by observing the measured disturbance tolerance values of the test cases, as described in Section 5.1.

6.1.1 Diagnostics test cases

As seen by the results detailed in Section 5.1.1, the frame Delay disturbance shows very stable results when applied to both the diagnostics test cases and the software update test. Since this disturbance adds a constant latency to the frames forwarded by the DITM, it is likely that the high enough disturbance magnitudes simply trigger timeouts in the tests, causing them to fail mostly deterministically. The deterministic nature of the disturbance can be seen as a limiting factor when testing the system using the diagnostic test cases, as the delay is likely to cause the test case to fail at the same points during each tolerance measurement iteration. This is also likely what causes the Delay disturbance to trigger so few unique errors, as seen in Section 5.2.

The Frame Drop and the Bit Error Rate tolerance results both show clear instability, due to their almost purely probabilistic nature. Due to this, the tolerances measured using these disturbances can only be considered approximate. However, predictability can be observed when comparing the Frame Drop tolerance results' mean and standard deviation values. These show that the expected deviation from the mean generally stays between 10-20%, no matter what the mean is. The Bit Error Rate,

meanwhile, does not share this property, showing greater variations from the mean.

The results for the Burst Timeout and Frame Reordering disturbances show great deviations from the mean in most cases, with little resemblance between them even though their implementations are similar. The Burst Queue disturbance, meanwhile, only seems to show the number of frames that can be stuck in the queue before the test cases' timeouts are triggered.

The greatest weakness of using these three disturbances in this context is likely twofold. With the rate of data flowing through the link being low, the disturbances' queues cannot fill with the number of frames that would challenge the receiving device's capability of quickly receiving and processing frames. When looking at the number of unique errors triggered by the disturbances in Figure 5.3, it becomes clear that the Burst Timeout and Frame Reordering disturbances act more like the Frame Drop and Bit Error Rate disturbances, by causing the test cases to fail at different points during execution, while the Burst Queue does not affect the tests in many varied ways.

Instead of using tests where the pass/fail verdict depends on only a few frames transmitted sparsely, the two burst disturbances require other testing methods to be properly evaluated. Instead, testing processes where the data rate is much higher will ensure that the limits of the device receiving the burst are tested more effectively. While the tolerances to the Frame Reordering disturbance are also very spread for the diagnostics tests, its worth is shown more for the software update test, as discussed in the next section.

6.1.2 Software update test

The tolerance results of the software update test show some different properties when compared to the diagnostics tests. The test case's Delay disturbance tolerance is lower but is approximately equally stable as when the disturbance is applied to the diagnostics tests. As mentioned in Section 5.1.2, the tolerance to Frame Loss for this test case is much lower and more stable when compared to the diagnostics test cases, which ultimately stems from the greater amount of traffic important to test case success generated by the software update and the choice of a disturbance quantum value of 1%. The current measurement indicates that the tolerance lies between 1.3% and 3.3%, but a lower quantum value would in this case allow for the tolerance to be measured with a higher level of granularity.

As described in Section 5.1.2, the disturbance logic of the Burst Timeout and the Frame Reordering had to be changed to allow the test case to succeed when using these disturbances, thereby allowing the proper measurement of the test case tolerance. This came about due to the same design limitation that caused the spread of the diagnostics test case burst tolerances; the threshold was checked every time a frame was intercepted by the DITM, causing the burst to send at skewed intervals. The change to the logic was applied and some tolerance measurements were run on the diagnostics test cases as well, but the results were similar to the original implementation and were therefore omitted from the results section.

When applied to the software update test case, the change to the logic helped clarify a point that was only approximately answered by the diagnostics tolerances, which is that the system does not suffer from the increased data rates of the burst disturbances when executing the test case functions. This is shown when comparing the Delay and Burst Timeout tolerances in Section 5.1.2, which indicates that the Burst Timeout causes the test case to fail by triggering a timeout in the test.

The comparison between the Burst Timeout and Frame Reordering tolerances indicates a much greater impact on the system than what could be reliably observed through the diagnostics test cases and reveals a relative weakness in the CS. It should be reminded, however, that this effect can only be properly observed on one test case, and can therefore not be confirmed as a system-wide effect.

In summary, the combined results of the disturbance tolerances show the following points:

- Comparing the patterns and results of the Delay and burst-based disturbances shows that the system does not respond unexpectedly due to higher data rates for the set of test cases used.
- Comparing the results of the Frame Loss and Bit Error Rate disturbance shows that while each individual result shows quite a large spread, the trends of the results are able to show higher and lower tolerances for different test cases. At the same time, this reinforces the need to run many tolerance measurements before the result can be trusted.
- The most prominent discovered weakness of CS may be Frame Reordering, but further testing on a greater number of functions is necessary to confirm this.

6.2 Deviations from expected system operation

The second research question was *Can the proposed testing method discover deviations from the expected operation of the system under test?*. This question is answered by analyzing the interactions noted in Section 5.3.

Two remarkably unexpected interactions were discovered during the tolerance measurements: the increased test execution time for test case F and the unresponsive VIU unit. Out of the two, only the increased test case runtime was entirely reproducible, shown with a very stable runtime increase in Figure 5.4 and less stable but occasionally even more increased runtime in Figure 5.5. While unexpected, the increase in runtime was also the only unexpected effect that the disturbances had on this test, as the tolerance showed comparable stability to the other test cases. This likely means that the cause of the interaction is an arbitrarily chosen timeout value for the kind of reset used for this test case, resulting in a very high worst-case scenario execution in the case of a total or near-total device disconnection. Reducing this timeout value would improve this worst-case execution time.

The second interaction is more notable, but also non-reproducible after the unresponsive VIU was replaced. While this may indicate that there is either a rare fault

in the software update process or a rare fault in the device hardware which manifested in the unresponsive device, the non-reproducible nature of the fault indicates that it is non-critical. With this in mind, investigation into possible causes for this interaction is advised. If such an effect manifests after e.g., a failed OTA update in a full vehicle, the result could in the worst case be the loss of all functions of the ECUs that are connected to the unresponsive VIU.

That only these two unexpected interactions were discovered during the tolerance measurements is not entirely surprising, due to the generally non-impactful effects of the test cases and the low number of available test cases to use for this project's experiments, but may mean that the ability of this method to find system deviations is low.

6.3 Benefits to the existing testing system

The third research question was *Can the proposed testing method benefit the verification and validation of the system under test?*. This question is answered by discussing the uses of the tolerance measurement results, the results showing the number of different errors triggered by the disturbances, and the DITM performance testing results.

The main benefit of the DITM implementation to the VTF testing process is the ability to automatically and repeatedly cause test cases to fail at different stages, and due to different disturbances, during execution. This kind of experimentation shows that test case failures due to failed assertions or faulty signals carry little risk of adversely affecting the system, while repeatedly showing what effects the disturbances do have, such as the observed interactions discussed in the previous section.

As seen in Figure 5.3, the implementation of the disturbances has a great effect on the number of different ways they can affect the system under test. This result shows that the disturbances that focus on the omission or corruption of data are better for disturbing this kind of stimuli in-depth, validating that the system can recover from different failures. While the Delay disturbance did not trigger as many errors, another merit to the disturbance is the validation that a function remains stable as long as the required frames arrive before the allocated timeout. As explained in Section 5.1.1.1, test case C does not show the same stability to this disturbance, instead generally failing at 1930ms as compared to the other tests' 1995ms, and showing one failure at 1280ms. This may indicate an instability in the operations carried out by the test case and warrants further investigation into the cause to rule out the possibility of the instability being connected to the method or the mode of the test case.

Due to the DITM system being logically separate from the VTF, the tolerance measurement process can be utilized on any function where data is transmitted between the VCU and the VIU, allowing the current implementation to be employed when e.g., ECUs are connected to the VIU as well. This also means that the tolerance measurement, and other disturbance testing, can be used without necessitating changes

to the VTF itself, meaning that the inclusion of the DITM system into the CS testing process does not affect how the VTF has to be maintained or designed. The automated nature of the approach also does not necessitate monitoring the tolerance testing while it is in progress.

While the test case stimuli did not challenge the DITM system in regards to throughput, the performance measurement results detailed in Section 5.4 show a better representation of the upper bounds of the throughput allowed by the DITM system. These results show that the DITM system is usable for experimenting with functions that generate traffic rates well above the rates of the ten test cases available for this project, showing that disturbance testing in a vehicle system with proper data rates is possible and should be pursued in future development of the method. While the DITM adds a noticeable latency to intercepted traffic, this latency is considered bounded and can therefore be accounted for during experimentation, although effort should be directed toward further optimizations of this latency.

6.4 Implementation considerations

This section discusses the DITM system and how different design choices can affect its usefulness in the TS and testing in a complete centralized vehicle network.

6.4.1 Tolerance search algorithm

As mentioned in Section 4.2.2.1, the tolerance search algorithm is based on the Galloping Search algorithm for searching sorted, unbounded lists, originally proposed by Bentley and Yao, as it grants efficiency in the number of test case executions needed to find its tolerance to a disturbance. In their original paper, Bentley and Yao compared the galloping approach to a unary search, where the search is conducted using constant increases as opposed to the exponential increase [31]. While such an approach to the tolerance measurement may give a similar performance to the exponential approach in some cases, it would depend on the choice of starting value, the choice of increment value, and the choice of quantum value. The efficiency of the exponential approach, meanwhile, only depends on the starting value and the quantum value.

The current implementation of the tolerance search algorithm is the same for all disturbances, and while it provides an efficient convergence time, optimizations can be applied to further increase this efficiency depending on the type of disturbance used. For instance, the Frame Loss disturbance, which uses a probability as its disturbance magnitude, is bounded by 100% probability. Therefore, finding an upper bound to the binary search is in this case not technically necessary, as the binary search can be executed using 100% as the upper bound, which would always be expected to cause the test case to fail, and 0% as the lower bound from the start.

Similarly, the Delay disturbance, which uses constant latency values, can benefit from either higher initial values or a greater factor of increase, such as $value * 4$ instead of $value * 2$ per iteration, which would speed up the discovery of the first

failure.

What is potentially lost by using this kind of optimization is the opportunity to observe system operation at low levels of disturbance magnitudes, which helps validate the overall stability of the function being tested and can catch failures that may arise at lower levels of disturbance, as observed with test case C in Table 5.1.

6.4.2 Hardware and software implementations

The throughput results in Figure 5.6 show that the Python, C, and Bridge Interface variants of the Forwarder Module implemented on the Workstation exceed the recommended data rate for both l values by some margin, and similar results are shown for the C and Bridge Interface approaches when implemented on the Raspberry Pi. Meanwhile, the Python implementation on the Raspberry Pi starts showing low amounts of packet loss at 15 Mb/s for $l = 64B$ and 30Mb/s for l=$128B$, making the margin for error much smaller than for the other implementations.

This likely means that to leverage the more portable nature of the Raspberry Pi, the DITM implementation has to use the C or Bridge Interface implementations, which come with some downsides.

Firstly, the opportunity to use a higher-level language like Python and the useful abstractions that come with it for disturbance implementation is lost, along with Python-specific tools like Scapy. Scapy allowed for the easy socket configuration used for the Python Traffic Forwarder implementation for this project and contains more abstractions and functions helpful for parsing and modifying frames, which can be useful for other disturbance implementations that target specific frame signatures. This would also make the DITM system less intuitive to engineers with no experience in lower-level languages.

Secondly, using the Bridge Interface implementation gives a worse opportunity to use user-defined functions for disturbance implementations, and disturbance implementations would therefore be limited to available network emulation tools like the tc-netem. This is not a problem for disturbances like delay and frame loss, but disturbances that require more control, like the Burst disturbances used for this project, cannot be implemented.

However, using the Raspberry Pi as the DITM comes with benefits to portability and energy consumption, as it is both smaller and uses less power than the Workstation variant. If implemented in a full-scale vehicle setup, where one DITM would be used between the VCU and each connected VIU, using Raspberry Pis would therefore be less resource-intensive as well as less difficult to transport and set up.

6.4.3 Performance measurements

As observed in Figure 5.6, the measured throughput of the Edge Device connected directly to the VCU via a Media Converter is shown to be generally equal to the throughput when using any DITM configuration, except for the Raspberry Pi with the Forwarder Module process assigned to processor 0. While this can be interpreted

as the performance being equal, a more likely explanation is that there are limiting factors in the hardware and software used during and for the measurement. Since the Media Converters are high-bandwidth and low-latency machines, the most likely culprit is the iPerf3 implementation on the VCU used for the measurement. The measurement can still properly investigate the Forwarder Module implementations' response to the data rates provided in Section 4.3.3 and the result was therefore accepted, but it leaves the question of the true upper throughput bounds unanswered.

6.5 Limitations

The main limitation is that the TS is not a fitting environment for testing Chaos Engineering in an automotive context, as the method specifies *production-like* environment. This is due to two major factors: the reduced number of functions available while using the test cases of the in-house test framework, and the low amount of data that these tests generate. While these test cases are highly repeatable, they don't allow for proper vehicle operation observation; instead, they only allow for observing the effects of diagnostics tests that have a low impact on the system.

Due to limited amounts of available hardware, the DITM hardware was only placed between the VCU and one VIU. To properly test the vehicle system, DITM units should be placed between the VCU and all connected VIUs. In the complete vehicle system, this would allow for fault-injection testing of all data flowing between the VCU and the VIUs, allowing for fault-injection testing of all vehicle functions without necessitating moving the hardware between the connections. Due to the test cases affecting all targeted devices in the same manner, implementing several DITM devices in the test rig would have given similar results and would not have greatly contributed to the tolerance measurements of this thesis.

6.6 Future work

This section describes the potential developments of this thesis project and how it should be implemented in a full-scale architecture.

The next step in the development and evaluation of the DITM system is to implement it in a setup more akin to a full-vehicle setting, such as a testing rig with all ECUs connected to the VIUs, and all VIUs being connected to the VCU through DITM hardware. This would provide the opportunity for experimentation under more production-like conditions, allowing for the Chaos Engineering method to be properly applied to the automotive software testing process.

In addition to this, investigation into using Scapy's frame-parsing functions to deconstruct intercepted frames and alter specific fields or parts of the payload should be conducted. This has the potential to allow disturbances to target specific functions in the data flow, increasing the fault mode coverage of the system.

7

Related work

7.1 Automotive fault injection

Fault injection in automotive software testing generally involves injecting faults into model simulations of the vehicle architecture, such as through Hardware-In-The-Loop simulations. In these approaches, signal delays, omission of data through packet loss, and bit-flip errors are often used to validate the system under test, along with more targeted faults such as offset, noise, and stuck-at faults, often targeting individual subsystems of the vehicle [32]–[35]. While this thesis uses several similar disturbance methods to these papers, this thesis focuses on developing a system capable of applying these disturbances to all data flowing between the units connected to the DITM, instead of purposefully isolating individual functions.

While using a physical device inserted into the Ethernet network for automotive fault injection is not common in the literature, the most prominent one found is the TRAITOR framework [36]. This framework functions via a fault injector implemented on a Zedboard FPGA development board for the Xilinx Zynq-7000 SoC, and is used to inject faults into Time-Triggered Ethernet (TTEthernet) connections. The framework uses a combination of byte insertion, frame delay, and frame jamming to emulate a number of TTEthernet faults, such as frame omission, corruption, device crashes, and several signal faults like stuck-at faults and changing signal IDs. TRAITOR has been used to successfully evaluate the TTEthernet networking technology in the automotive domain, using the corruption failure mode to investigate how latency is affected by this disturbance [37]. The hardware-based method for fault injection and the fault modes used are similar to the DITM approach and the disturbances developed for this thesis. This project's DITM approach focuses more on the framework's extensibility through the use of Python, a high-level and generally easy-to-understand programming language, for the disturbance implementation, which comes with performance trade-offs as discussed in Section 6.4.2. Additionally, the DITM approach explores interfacing with existing test frameworks to observe system response and seeks to quantify the system's tolerances to the implemented disturbances.

7.2 Chaos Engineering framework architecture

No applications of the Chaos Engineering method to the automotive domain have been found during the research for this project. To inspire the DITM software architecture, the methods that several successful Chaos Engineering frameworks use to interface with and affect their target systems were investigated. The most influential of these were the ChaosXploit framework [18], the CloudStrike framework [17], and the Phoebe framework [38]. While the details of their targeted systems, experimentation goals, and methods differ, they all incorporate similar system modules. Common modules include a module to coordinate the fault injection (this project's Controller module), a module for monitoring the effects of the experiment (Analysis module), and a module for injecting faults (Traffic Forwarder module). Like this project, Pheobe also uses a visualizer to visualize the results of their experiments.

8

Conclusion

The move towards centralized automotive architectures comes with notable benefits to scalability, communication performance, and the number of ECUs needed for vehicle function. Simultaneously, it introduces a single point of failure in the internal communication system, requiring diligent investigation into how such a failure may affect the vehicle system as a whole.

In this thesis, a method for negative testing at the link layer of the internal communication architecture was introduced to pave the way for Chaos Engineering - negative testing in a production-like automotive environment. By developing a Device-In-The-Middle (DITM) system and applying this to the existing Test System at Volvo Cars, disturbing the communication between the vehicle's control unit (VCU) and its connected gateway units (VIU), the research questions of this thesis were answered.

Q1: Can the proposed testing method reliably measure and validate the system under tests tolerance to communication disturbances?

Several approximate disturbance tolerances were observed by analyzing the results gathered by applying the tolerance search algorithm to the Test System setup. For the Delay disturbance, the tolerances were very stable for the majority of the test cases, while one test case showed some instability. For the Frame Loss and Bit Error Rate disturbances, some spread was observed in the tolerance results. However. the same trends for the test cases were observed for both disturbances, where some test cases showed repeatedly lower resilience than others. The Burst disturbance results showed a significant spread when applied to the first nine test cases, even after alteration to the implementation, but were more accurate when applied to the software update test, whose results showed that the Core System does not suffer from increased data rates at the data volumes that the test cases induce. Another disturbance that the system showed resilience to is the Frame Duplication, which never caused a test failure. The Frame Reordering disturbance tolerance was shown to be very low when applied to the software update process, indicating a vulnerability.

Q2: Can the proposed testing method discover deviations from the expected operation of the system under test?

One instance of vastly increased test case execution time was observed for very high disturbance magnitudes for several of the disturbances. This is likely connected to an arbitrarily set timeout value for a reset used during this function, and reducing this

value would provide a worst-case optimization for the test case. After disturbing the software update test case, a VIU unit was left unresponsive and unrecoverable. Since this effect was not observed when the VIU was replaced, it may indicate a rare fault that should be investigated further, to avoid such an effect should the update process fail in a vehicle. These interactions speak to the disturbance solution's ability to discover some inefficient interactions and unintended effects, but the solution should be applied to a wider range of functions for further evaluation and development.

Q3: Can the proposed testing method benefit the verification and validation of the system under test?

The method allows for automatic repeated test case execution under unpredictable conditions, validating the Test System's ability to repeatedly recover from faulty function executions at different execution stages. The differing tolerance measurements gained from applying the disturbances to the test cases help show where some instabilities may lie, such as with some Delay and frame reordering results, and which test cases are generally more vulnerable to data omission. The results also validate the Test System's resistance toward increased data rates and frame duplication. Performance testing shows that the method has the potential to scale to much higher data rates than generated by the test cases, and the latency incurred by the DITM can be bounded and accounted for.

All in all, the DITM method shows promise when applied to the Test System, implying that with some development, the method can be used to introduce Chaos Engineering-based experimentation into the validation and verification of centralized automotive architectures. Further research should focus on the application of the method in a more complete vehicle setup, for instance, a deconstructed system with all ECUs connected, to further develop the method and properly evaluate its use in this environment.

Bibliography

[1] I. Harris, "Chapter 22 - embedded software for automotive applications," in *Software Engineering for Embedded Systems*, R. Oshana and M. Kraeling, Eds., Oxford: Newnes, 2013, pp. 767–816, ISBN: 978-0-12-415917-4. DOI: https://doi.org/10.1016/B978-0-12-415917-4.00022-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124159174000220.

[2] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, 2007. DOI: 10.1109/JPROC.2006.888386.

[3] R. Isermann, R. Schwarz, and S. Stolzl, "Fault-tolerant drive-by-wire systems," *IEEE Control Systems Magazine*, vol. 22, no. 5, pp. 64–81, 2002. DOI: 10.1109/MCS.2002.1035218.

[4] I. Salman, B. Turhan, and S. Vegas, "A controlled experiment on time pressure and confirmation bias in functional software testing," *Empirical Software Engineering*, vol. 24, 2019. DOI: https://doi.org/10.1007/s10664-018-9668-8.

[5] I. Hooda and R. S. Chhillar, "Software test process, testing types and techniques," *International Journal of Computer Applications*, vol. 111, no. 13, 2015.

[6] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, Feb. 2016, ISSN: 0360-0300. DOI: 10.1145/2841425. [Online]. Available: https://doi.org/10.1145/2841425.

[7] ISO: International Organization for Standardization. "Iso 26262-1:2018." (), [Online]. Available: https://www.iso.org/standard/68383.html (visited on 03/05/2024).

[8] J. Gustavsson. "Why automotive software needs more rust." (2023), [Online]. Available: https://fileadmin.cs.lth.se/cs/Education/EDAN65/2023/lectures/L13-vcc_rust_lth_231003.pdf (visited on 04/22/2024).

[9] unknown. "White paper : Centralizing compute: How far can it go?" (2022), [Online]. Available: https://www.aptiv.com/docs/default-source/white-papers/2022_aptiv_whitepaper_centralizingcomputesoftware.pdf?sfvrsn=b0c21238_4/2022_Aptiv_WhitePaper_CentralizingComputeSoftware (visited on 04/22/2024).

[10] unknown. "Microchip: Central compute." (unknown), [Online]. Available: https://www.microchip.com/en-us/solutions/automotive-and-transportation/adas-and-av/central-compute (visited on 04/22/2024).

[11] unknown. "Nxp: Central compute." (unknown), [Online]. Available: https://www.nxp.com/applications/automotive/software-defined-vehicle/central-compute:CENTRAL-COMPUTE (visited on 04/22/2024).

[12] V. Bandur, G. Selim, V. Pantelic, and M. Lawford, "Making the case for centralized automotive e/e architectures," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 2, pp. 1230–1245, 2021. DOI: 10.1109/TVT.2021.3054934.

[13] M. Kumar, B. Achutan, and S. Badade, "Centralized e/e architecture and evolution," *SAE Technical Papers*, 2023. DOI: https://doi.org/10.4271/2023-01-1511.

[14] L. Mauser, S. Wagner, and P. Ziegler, "Methodical approach for centralization evaluation of modern automotive e/e architectures," in *Software Architecture. ECSA 2022 Tracks and Workshops*, T. Batista, T. Bure, C. Raibulet, and H. Muccini, Eds., Cham: Springer International Publishing, 2023, pp. 165–179, ISBN: 978-3-031-36889-9.

[15] A. Basiri, N. Behnam, R. de Rooij, *et al.*, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016. DOI: 10.1109/MS.2016.60.

[16] Netflix, *Chaos monkey*, 2016. [Online]. Available: https://github.com/Netflix/chaosmonkey.

[17] K. A. Torkura, M. I. H. Sukmana, F. Cheng, and C. Meinel, "Cloudstrike: Chaos engineering for security and resiliency in cloud infrastructure," *IEEE Access*, vol. 8, pp. 123 044–123 060, 2020. DOI: 10.1109/ACCESS.2020.3007338.

[18] S. Palacios Chavarro, P. Nespoli, D. Díaz-López, and Y. Niño Roa, "On the way to automatic exploitation of vulnerabilities and validation of systems security through security chaos engineering," *Big Data and Cognitive Computing*, vol. 7, no. 1, 2023, ISSN: 2504-2289. DOI: 10.3390/bdcc7010001. [Online]. Available: https://www.mdpi.com/2504-2289/7/1/1.

[19] H. H. Bengtsson, M. Hiller, and S. Sigfridsson, *Tsn ethernet as core network in the centralized e/e architecture - challenges and possible solution*, 2019. [Online]. Available: https://standards.ieee.org/wp-content/uploads/import/documents/other/eipatd-presentations/2019/D1-02_BENGTSSON-TSN_ethernet_as_core_network_in_EE_architecture.pdf.

[20] esnet, *Iperf*, https://github.com/esnet/iperf, 2014.

[21] F. L. Stephen Hemminger and H. P. Pfeifer. "Archlinux tc-netem manual page." (2001), [Online]. Available: https://man.archlinux.org/man/core/iproute2/tc-netem.8.en (visited on 04/22/2024).

[22] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 595–612.

[23] X. Networks. "Microbursts, jitter, and buffers." (), [Online]. Available: https://www.xenanetworks.com/wp-content/uploads/xenadocuments/Microburst-WP.pdf.

[24] F. Fummi, D. Quaglia, and F. Stefanni, "Network fault model for dependability assessment of networked embedded systems," Oct. 2008, pp. 54–62. DOI: 10.1109/DFT.2008.21.

[25] Philippe Biondi, *Scapy documentation*, 2024. [Online]. Available: https://scapy.readthedocs.io/en/latest/introduction.html.

[26] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[27] Wikipedia contributors, *Binary search algorithm*, [Online; accessed 20-05-2024], 2024. [Online]. Available: https://en.wikipedia.org/wiki/Binary_search_algorithm.

[28] R. Baeza-Yates and A. Salinger, "Fast intersection algorithms for sorted sequences," in *Algorithms and Applications: Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday*, T. Elomaa, H. Mannila, and P. Orponen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 45–61, ISBN: 978-3-642-12476-1. DOI: 10.1007/978-3-642-12476-1_3. [Online]. Available: https://doi.org/10.1007/978-3-642-12476-1_3.

[29] H. Guissouma, C. P. Hohl, F. Lesniak, M. Schindewolf, J. Becker, and E. Sax, "Lifecycle management of automotive safety-critical over the air updates: A systems approach," *IEEE Access*, vol. 10, pp. 57 696–57 717, 2022. DOI: 10.1109/ACCESS.2022.3176879.

[30] A. Nagpal and G. Gabrani, "Python for data analytics, scientific and technical applications," in *2019 Amity International Conference on Artificial Intelligence (AICAI)*, 2019, pp. 140–145. DOI: 10.1109/AICAI.2019.8701341.

[31] J. L. Bentley and A. C.-C. Yao, "An almost optimal algorithm for unbounded searching," *Information processing letters*, vol. 5, no. SLAC-PUB-1679, 1976.

[32] M. Abboush, D. Bamal, C. Knieke, and A. Rausch, "Hardware-in-the-loop-based real-time fault injection framework for dynamic behavior analysis of automotive software systems," *Sensors*, vol. 22, no. 4, 2022, ISSN: 1424-8220. DOI: 10.3390/s22041360. [Online]. Available: https://www.mdpi.com/1424-8220/22/4/1360.

[33] M. Reorda and M. Violante, "Hardware-in-the-loop-based dependability analysis of automotive systems," in *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, 2006, 6 pp.-. DOI: 10.1109/IOLTS.2006.41.

[34] R. Svenningsson, H. Eriksson, J. Vinter, and M. Törngren, "Model-implemented fault injection for hardware fault simulation," in *2010 Workshop on Model-Driven Engineering, Verification, and Validation*, 2010, pp. 31–36. DOI: 10.1109/MoDeVVa.2010.11.

[35] P. Folkesson, F. Ayatolahi, B. Sangchoolie, J. Vinter, M. Islam, and J. Karlsson, "Back-to-back fault injection testing in model-based development," in *Computer Safety, Reliability, and Security*, F. Koornneef and C. van Gulijk, Eds., Cham: Springer International Publishing, 2015, pp. 135–148, ISBN: 978-3-319-24255-2.

[36] D. Onwuchekwa, "Fault injection framework for time-triggered systems," Ph.D. dissertation, Universität Siegen, 2020. DOI: http://dx.doi.org/10.25819/ubsi/5828. [Online]. Available: https://dspace.ub.uni-siegen.de/handle/ubsi/1727.

[37] D. Onwuchekwa, S. Mittal, and R. Obermaisser, "Evaluating ttethernet in the automotive domain using fault injection," in *AmE 2022 - Automotive meets Electronics; 13. GMM-Symposium*, 2022, pp. 1–6.

[38] L. Zhang, B. Morin, B. Baudry, and M. Monperrus, "Maximizing error injection realism for chaos engineering with system calls," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2695–2708, 2022. DOI: 10.1109/TDSC.2021.3069715.